

Recherche de similarités dans des codes sources

Projet libre de spécialité SCIA à l'EPITA.
Responsable Akli Adjaoute.

Marco TESSARI Jérôme POUILLER

EPITA - Octobre 2004

Table des matières

1	Introduction	4
1.1	Problématique des assistants	4
1.2	Extensions aux problématiques extérieures	4
1.3	Cas d'utilisations	5
1.4	Stratégie commerciale du projet	5
2	Étude de la problématique	6
2.1	Valeur judiciaire des résultats de notre logiciel	6
2.2	Projet général	6
2.3	42sh	7
2.4	Tiger	7
2.5	Qu'est ce que la triche ?	7
2.6	Autre types de triches	8
2.7	Contraintes	9
2.7.1	Résultats	9
2.7.2	Contraintes de temps	9
2.7.3	Contraintes d'implémentation	9
2.7.4	Recentrage des rendus	9
3	État de l'art	10
3.1	Étude des fichiers objets	10
3.2	Reconnaitances des erreurs	10
3.3	Ctcompare (Trichotomie)	10
3.4	Simian	11
3.5	CodeMatch	11
3.6	MyDropBox	11
3.7	Cogger[5]	11
3.8	Sherlock	12
3.9	Moss[14]	12
3.10	Yap3[19]	12
3.11	Outils externes utiles	12
3.11.1	GNU C Compiler (gcc)	12
3.11.2	Object Dumper (objdump)	12
3.11.3	strip	12
3.11.4	strace	13
3.11.5	GNU Profiler (gprof)	13

3.11.6	ctags et etags	13
3.11.7	indent	13
3.11.8	wdiff et diff	13
4	Solutions retenues	15
4.1	Empreinte de fichier	15
4.1.1	Trace d'exécution de programme	15
4.2	Distance entre les fichiers	15
4.2.1	Distance sur les mots rares ¹	16
4.2.2	La distance binaire	16
4.2.3	La plus longue sous-chaîne (Longest Common Substring, LCS)	16
4.2.4	Distance de Levenshtein	16
4.2.5	Greedy String Tiling (GST)	17
4.3	Optimisations	19
4.3.1	Clés MD5	19
4.3.2	Programmation dynamique	19
4.3.3	Karp-Rabin	20
4.3.4	Greedy String Tiling avec Karp-Rabin	20
4.3.5	Compression de tokens	22
4.3.6	Utilisation du cache	22
4.4	Fichiers à comparer	22
4.4.1	Analyse syntaxique	25
4.4.2	Analyse sémantique	26
4.4.3	Division en fonction	29
4.5	Conclusion	29
4.6	Les techniques "sioux"	29
4.6.1	Recherche d'incohérences	30
4.6.2	Recherche des relations sociales	30
4.6.3	Surveillance des comptes	30
4.6.4	Surveillance de la commande scp	30
4.6.5	L'obscurantisme	31
4.6.6	La psychologie	31
4.7	Expression du résultat et Techniques d'interprétation	31
4.7.1	Datamining	31
4.7.2	Système à base d'apprentissage unique (type Réseaux de Neurones)	31
4.7.3	CBR	32
4.7.4	Système expert	32
5	Implémentation	33
5.1	Plateforme de développement : TestTKewl	33
5.1.1	Récapitulatif des besoins	33
5.1.2	Présentation de l'outil	34
5.1.3	Architecture du projet	36
5.1.4	Ecriture d'un test	38
5.1.5	Langage d'entrée	39

¹Merci à Akim Demaille pour cette idée

5.1.6	Le filtrage	40
5.1.7	Autres particularités	41
5.1.8	Chaîne d'utilisation	43
5.1.9	Bilan	44
A	Spécifications de la TestKewl	46
A.1	Description générale du projet	46
A.1.1	Buts	46
A.1.2	Principaux cas d'utilisation de la moulinette	46
A.2	Spécifications générales	47
A.2.1	Fonctionnement général	47
A.2.2	Méthode devant apparaître de base dans la moulinette	48
A.2.3	Cas pratique	49
A.2.4	Modules de sortie	50
A.2.5	Mise en place d'un processus de qualité	50
A.2.6	Coding Style	50
A.2.7	Erreurs	51
A.2.8	Execution de scripts extérieurs	51
A.2.9	Interface avec l'extérieur	52
A.2.10	Options de la ligne de commande	52
A.2.11	Création automatique des références	52
A.2.12	Autres détails et bugs à éviter	53
A.3	Interfaçage avec l'intranet	53
A.3.1	Xml	53
A.3.2	XSL	55
A.4	Glossaire	55
	Références	57
	Index	57

Table des figures

3.1	Exemple de sortie de <code>ctcompare</code>	11
3.2	Exemple de graphe de sortie de <code>ctcompare</code>	11
3.3	Exemple de sortie de <code>gprof</code>	14
4.1	Implémentation naïve de la distance de Levenshtein	17
4.2	Implémentation naïve de la distance de GST	18
4.3	Optimisation de la distance de Levenshtein par programmation dynamique	19
4.4	Implémentation naïve d'une sous-chaîne dans une chaîne	20
4.5	Optimisation à l'aide d'une fonction de hash	20
4.6	Algorithme de Karp-Rabin (en C)	21
4.7	Optimisation simple sur la distance de GST (en C)	23
4.8	Suite de 4.7	24
4.9	Différences sémantiques	24
4.10	Différences sémantiques	25
4.11	Exemple de tokenisation	26
4.12	Exemple de code sémantiquement identiques	27
4.13	Résultat compilé de codes sémantiquement identiques	28
5.1	Détails des différentes étapes de <code>TestTKewl</code>	37
5.2	Graphe résultat de la compilation de règles d'un filtre	42
5.3	Détails de la chaîne de notation des assistants.	44
A.1	Exemple d'UML de classe de la partie Test de la moulinette	47

Notes importantes

Clause de discrétion

Comme nous le verrons par la suite, l'obscurantisme est un élément important de la recherche de cas de triche. Pour cette raison, nous posons quelques restrictions sur la publication de ce rapport. Notre principal problème est qu'il ne soit pas divulgué aux élèves de première année d'ingénierie d'EPITA. On peut considérer que les risques qu'un élève fasse des recherches de manière à trouver ce document sans qu'on lui en donne l'idée sont minime. Nous ne posons aucune clause de confidentialité, mais, nous demandons à tous les possesseurs et lecteurs de cet article une certaine discrétion sur son existence et son contenu.

Public visé

Ce rapport se destine en particulier aux Assistants d'EPITA. Toutefois, nous nous efforcerons de généraliser le contenu aux problématiques extérieures à EPITA.

Ce rapport ne se contente pas de proposer une solution. Il s'efforce de montrer toutes les voies que nous avons suivies et les résultats que nous avons obtenus dans le but que ce travail ne soit pas redondant.

Chapitre 1

Introduction

Sommaire

1.1 Problématique des assistants	4
1.2 Extensions aux problématiques extérieures	4
1.3 Cas d'utilisations	5
1.4 Stratégie commerciale du projet	5

L E BUT de ce projet est d'étudier la problématique du repérage des cas de vol de codes sources.

1.1 Problématique des assistants

Lors du cycle ingénierie à EPITA, les assistants jouent un rôle pédagogique important. Ils gèrent les principaux projets des nouveaux étudiants à EPITA.

Les Assistants sont une vingtaine et doivent gérer environ une trentaine de projets et une dizaine d'exams machines¹.

Les projets sont fait en groupes de 1 à 6. Sur une année, les assistants corrigent environ 9000 rendus ! Ce travail ne serait pas possible sans l'aide des moulinettes. Une moulinette est un programme créé par les assistants pour automatiser la correction d'un projet. Il existe plusieurs sortes de moulinettes. La moulinette de correction est la plus utilisée.

Dans une promotion de 300 personnes, les cas de triches et d'inspiration de code d'un tiers sont relativement courants et difficilement repérable à cause du nombre de rendu et de possibilités. Ce traitement est irréalisable par un humain. Pour le moment, il existe plusieurs solutions mais aucune n'est optimale ni satisfaisante.

Le but de notre projet est de créer une moulinette de triche. C'est à dire un programme capable de retrouver parmi la promotion les personnes ayant recopié du code, s'étant inspirées du code d'un autre ou encore les personnes ayant réalisé le projet à plusieurs.

¹12 exams machines, 12 jours de piscine, fnmatch, evalex, libstream, find, malloc, tron, projetx, ftpd, minishell, threads, asm, bistromathique, corewar, 5 jours de piscine C++, chess, T0, T1, T2, T3, T4, T5, 42sh, dromadaire, biblio, Zelda, Projet de graph, etc...

1.2 Extensions aux problématiques extérieures

Ce type de projet est particulièrement recherché en ce moment. Effectivement, le copier coller sauvage de code n'est une problématique interne à EPITA. Dans l'industrie, on trouve souvent des cas d'espionnage industriel et de copie de code. L'expansion rapide de l'Open Source ces dernières années rend la pratique encore plus aisée. Généralement, les projets Open Sources permettent à un tiers de réutiliser le code à condition que le nouveau code soit gratuit et Open Source à son tour. Il arrive souvent que des industriels peu scrupuleux reprennent des codes Open-Source pour des projets à but commerciaux. Cette pratique est absolument illégale, mais, il est souvent très difficile de prouver le vol vu que la victime n'a pas accès au code de du voleur. Le cas de Sco Linux l'année dernière à fait beaucoup de bruit [15].

On remarque aussi les vols de sources sont de plus en plus fréquents. Par exemple, le vol des sources de Microsoft [12] cette année et ceux de Quake [2] l'année dernière. Les codes ainsi divulgués ont pu être repris par beaucoup de programmeurs.

Ajoutons enfin l'ouverture des codes de nombreux industriels, tels que Microsoft [7]. Ces codes ne sont généralement pas sous Licence Open-Source. Ils sont divulgués à un public relativement restreint dans le but de permettre le développement efficace d'applications compatibles. Les industriels aimeraient se passer de tout ce processus et avoir la possibilité d'ouvrir leur code pour que les concurrents fassent des logiciels compatibles, mais, ils ne veulent surtout pas donner d'idées².

Pour tout ces cas de figures, la détection de copies de codes frauduleuses permettrait de limiter les abus. Le but n'est alors pas de détecter tous les cas. Il suffit d'en détecter 1 sur 10 pour que le procédé soit suffisamment dissuasif.

Contrairement à la problématique de la triche à EPITA, la copie de code industriel oblige à travailler sur des programmes compilés. On doit donc pour satisfaire cette problématique être capable de donner des résultats sur des binaires ou des code assembleurs.

Si un utilitaire tel que le notre voyait le jour, il faudrait connaître quelle valeur judiciaire il aurait. C'est la question que nous avons étudié dans le chapitre 2.1.

1.3 Cas d'utilisations

Notre principal but est de résoudre la problématique des assistants³. Néanmoins si nous arrivions à proposer une solution applicable à la problématique du vol de source au niveau industriel, ça serait un plus pour notre projet.

1.4 Stratégie commerciale du projet

En étudiant la problématique de l'Open Source, nous avons trouver que le meilleur moyen de vendre notre projet était, paradoxalement, de le placer sous licence GPL (gratuit). Effectivement, même si le projet est très recherché, nous ne pourrions pas le vendre sans avoir une enseigne de confiance devant nous. Cela signifierait le vendre à une entreprise renommée ou bien créer une entreprise. Au lieu de cela, nous préférons donner le projet. Les industriels intéressés le testeront et ils se feront une idée par eux-mêmes de la qualité du logiciel. Les noms des auteurs du projets

²C'est ce qui s'appelle vouloir le beurre et l'argent du beurre

³C'est-à-dire la détection de la triche

ne serons alors pas oublié. Nous préférons donc miser sur l'avenir et notre carrière professionnel plutôt que de voir une rémunération (naïve) à court terme.

Chapitre 2

Étude de la problématique

Sommaire

2.1	Valeur judiciaire des résultats de notre logiciel	6
2.2	Projet général	6
2.3	42sh	7
2.4	Tiger	7
2.5	Qu'est ce que la triche ?	7
2.6	Autre types de triches	8
2.7	Contraintes	9
2.7.1	Résultats	9
2.7.2	Contraintes de temps	9
2.7.3	Contraintes d'implémentation	9
2.7.4	Recentrage des rendus	9

NOUS avons tous d'abord commencé par étudier le problème en profondeur et à lister exhaustivement les cas d'utilisation de notre projet.

2.1 Valeur judiciaire des résultats de notre logiciel

2.2 Projet général

D'une manière générale, toute la promotion reçoit le même sujet où les informations sur le fonctionnement du programme sont toujours très précise. Les élèves partent toujours de rien pour coder leurs projets, mais, ils reprennent souvent du code qu'ils ont déjà écrit pour un projet précédent.

Les élèves rendent leur travail sous la forme d'une tarball, c'est à dire d'une archive compressée contenant tout leur travail¹.

Nous devons analyser jusqu'à 350 rendus (lorsque les projets se font seul). Le tableau 5.1 donne une idée du nombre de lignes de codes de chaque projet.

¹Principalement du code

TAB. 2.1 – Nombre de lignes de code moyen par projet

Projet	Nombre de lignes	Nombre de personnes par groupe
Jour 2 de Piscine	277	1
Bistromathique	1481	2
Minishell	852	1
Corewar	4869	4
Chess	965	2
Minimake	427	1
42sh	19579	6
Tiger	25590	4

Nous verrons le cas du projet Tiger dans la section 2.4

On remarque que le pire projet est 42sh avec presque 20000 lignes de code pour 6 personnes, soit environ 3000 lignes par personne. Ce projet est celui impliquant le plus de lignes. Il y a un fort écart entre 42sh et la moyenne des projets de l'année. On remarque que les projets dépassent rarement les 1000 lignes de code par personne.

2.3 42sh

42sh est des projets les plus gros avec presque 2000 lignes de code pour 6 étudiants (cf. tableau 5.1). Néanmoins, le projet est divisé en 4 rendus. A chaque rendu, les élèves nous donnent un tarball contenant leur travail. On remarque que la triche devient de plus en plus complexe au fur et à mesure que le projet avance. Effectivement, chaque groupe choisit une modélisation légèrement différente et il devient difficile de trouver un code facilement réutilisable pour sa propre version². Par conséquent, l'application de système anti-triche sur un projet possédant plus de 1000 lignes de code par élève devient facultative.

2.4 Tiger

Un des projets où les cas de triches sont les plus complexes à retrouver est *Tiger*. Tiger est un projet de 6 mois divisé en un certain nombre de tranches. A chaque tranche, une nouvelle étape du projet est demandé à l'élève. A chaque tranche, les assistants fournissent du code aux élèves. Les intérêts sont multiples :

- Il permet aux étudiant de n'écrire que le code intéressant du projet
- Il permet de faire travailler les étudiants sur un gros projet tel qu'ils le feront dans leur vie professionnelle (Le tableau 5.1 montre que le projet contient plus de 25000 lignes de codes)
- Il aborde le problème de la lecture du code d'un autre codeur
- Le code donné sera lu par les étudiants, il est spécialement fait pour qu'ils en tirent des bonnes habitudes.

Ce projet est différent des autres par le fait que du code est donné et surtout que le *même* code est donné à tous le monde. Par conséquent, les rendus des élèves se ressemblent énormément.

²Ou alors, le code demande une forte adaptation ce qui implique de le comprendre

Néanmoins, on connaît quel est le code fourni. Il est donc possible d'extraire du code rendu le code effectivement modifié par les élèves.

2.5 Qu'est ce que la triche ?

Nous parlons depuis le début de *triche*. Nous devons à notre niveau définir de manière plus précise les différents types de triche et leur gravité. Effectivement, il est important de faire la différence entre triche et entraide des élèves. Il est important pour nous de ne pas tuer l'esprit d'entraide entre les élèves. Effectivement, c'est le points fort de l'enseignement à EPITA. L'entraide permet aux étudiants de mieux comprendre certaines notions. Nous devons donc faire très attention à ne pas être trop dur dans nos résultats.

1. Tout d'abord, considérons un étudiant qui demande le fonctionnement d'un point précis à un camarade. Le code résultant de cette réponse risque d'être identique dans les rendu des deux étudiants. Nous ne pouvons néanmoins pas considéré ceux-ci comme de la triche, mais, comme de l'entraide.
2. Ensuite, il arrive très souvent que les élèves parlent entre eux du projet et de la manière de le coder ou l'architecturer. Il germe généralement de ces discussions des nouvelles idées très constructives et permettent aux élèves de réfléchir aux problématiques qui leurs sont fournies. Néanmoins, les programmes écrit par les deux élèves risquent de beaucoup se ressembler. Nous devons donc faire attention à ne pas déclarer ses étudiants comme "tricheurs".
3. Il arrive qu'une personne explique un principe algorithmique à une autre en lui fournissant éventuellement la fonction dont il a besoin. Dans ce cas, le jugement est difficile. Notre but est que l'élève apprenne des choses et il probable que l'étude du code de l'un de ces camarades lui soit bénéfique. Néanmoins, si le code est repris sans qu'il en connaisse tout le sens, on peut alors considérer qu'il y a un cas d'inspiration de code voir de triche. Nous devons détecter ces cas. Seul un humain pourra ensuite dire si l'élève a parfaitement compris le code qu'il a écrit ou si il s'est contenté d'appliquer le même algorithme bêtement. La gravité dépend alors de la quantité de code non compris.
4. Enfin, on voit parfois des cas de recopiage pur et dur ou des fichiers entiers sont volé. Ces pratiques sont absolument prohibées.

Nous avons ici pris des exemples de cas directs. Il arrive bien souvent que la ressemblance de code soit indirecte. Par exemple, lorsque deux élèves recherche des informations sur la même page internet. Dans ces cas, les mêmes règles s'appliquent. On peut s'inspirer du code à condition d'en connaître le sens. Le copier coller idiot depuis Internet est interdit.

2.6 Autre types de triches

A part la copie et l'inspiration de code. Nous avons aussi réfléchis aux autres types de triches pouvant être tentées par les élèves. Les élèves pourraient par exemple voler les sources des solutions de références codées par les assistants. Ce type de triche pourra être facilement détectée par notre projet en intégrant les sources des références à notre base de rendus.

Parmi les autres types de triche, lors du passage de la moulinette de correction pour les projets et plus particulièrement sur les projets à plusieurs tranches (Tiger et 42sh), un étudiant peut exécuter

du code malicieux. Par exemple, il peut s'envoyer un mail contenant les tests de la moulinette de correction, ou alors placer une backdoor sur le compte des assistants. Par exemple, ajouter son login dans le fichier `.forward` à la racine du compte lui permettrait de recevoir les mails des assistants³. Pour ce genre de triche, il suffit de repérer certains appels de fonctions systèmes et de prévenir les assistants lorsqu'un appel à une fonction interdite ce produit. On peut facilement faire ce genre de manipulation grâce à la variable d'environnement `$LD_PRELOAD` dans les environnements Unix. Cette variable permet de demander le chargement de certaines bibliothèques en priorité et on peut ainsi détourner certains appels de fonctions vers des fonctions codées par les assistants.

2.7 Contraintes

2.7.1 Résultats

Les résultats du projet n'ont pas besoins d'être parfaits. Effectivement, le principal but est dissuasif. Laisser passer 50% des tricheurs est satisfaisant. De même, on peut se permettre de noter des personnes tricheuses alors qu'elles ne le sont pas. Un Assistant vérifiera tous les cas de triches et surtout les élèves sont convoqués pour s'expliquer.

Néanmoins, on veut que toutes les formes de triches soient repérées. Il ne faut qu'il existe un moyen de contourner la moulinette. De plus, le système doit être le plus automatisé possible. Nous n'avons pas le temps d'étudier les cas de triches en profondeur⁴. Le système doit permettre de repérer automatiquement les cas de triche et d'avertir les assistants.

2.7.2 Contraintes de temps

Pour un tel projet, le temps est un facteur déterminant. Comme nous l'avons souligné dans la section 2.2, nous devons analyser environ 350 rendus chacun inférieur à 1000 lignes. L'analyse d'un projet ne devrait pas prendre plus de 24 heures pour que l'utilisation soit confortable. Néanmoins, EPITA possède beaucoup d'ordinateur. Nous pouvons utiliser les machines du réseau en parallèle. Nous devons simplement respecter les utilisateurs et ne pas faire fonctionner la moulinette sur une machine ou travaille quelqu'un.

2.7.3 Contraintes d'implémentation

Les machines de l'école fonctionnant sous NetBSD, notre projet doit pouvoir fonctionner sous cette architecture.

2.7.4 Recentrage des rendus

On remarque aussi que nous n'avons pas besoin de "recentrer" les fichiers d'entrées. Effectivement, tous les rendus traités proviennent du même projet. Nous n'avons donc pas d'analyses préliminaires à faire pour supprimer les sources des programmes n'ayant rien à voir (Ça serait le cas si nous devions travailler sur des résultats d'Internet par exemple...)

³Cela ne fonctionnerait pas en réalité vu que les corrections des élèves sont exécutées sur un compte isolé du reste de l'environnement des assistants

⁴Ça n'est pas notre travail

Chapitre 3

État de l’art

Sommaire

3.1 Étude des fichiers objets	10
3.2 Reconnaissances des erreurs	10
3.3 Ctcompare (Trichotomie)	10
3.4 Simian	11
3.5 CodeMatch	11
3.6 MyDropBox	11
3.7 Cogger[5]	11
3.8 Sherlock	12
3.9 Moss[14]	12
3.10 Yap3[19]	12
3.11 Outils externes utiles	12
3.11.1 GNU C Compiler (gcc)	12
3.11.2 Object Dumper (objdump)	12
3.11.3 strip	12
3.11.4 strace	13
3.11.5 GNU Profiler (gprof)	13
3.11.6 ctags et etags	13
3.11.7 indent	13
3.11.8 wdiff et diff	13

NOUS nous sommes tout d’abord intéressé aux programme existants dans ce domaine.

3.1 Étude des fichiers objets

Cette technique était utilisée par les Assistants 2004. Elle consiste tout simplement à compiler tous les fichiers de tous les étudiants avec le même compilateur et les mêmes options (sans optimisation, sans debugging). Puis, on *strip*¹ les fichiers objets et enfin on compare tous les fichiers de tous les rendus.

¹Processus consistant à retirer tous les symboles inutiles d’un fichier objet

FIG. 3.1 – Exemple de sortie de `ctcompare`. On voit ici que les identifiants ont été remplacés par des identifiants généraux.

```
18 login_1/regalloc1.c:3-3 login_2/regalloc2.c:4-3
;
for ( id105 = NUM0 ; id115 && id115 [ id105 ] ; ++ id105 ) ;
```

FIG. 3.2 – Exemple de graphe obtenu à l'aide `dotty` et de `ctcompare`.

Ce système permet de repérer les copies de fichiers. Puisqu'elle étudie les fichiers compilés, elle permet de passer outre les noms de variables, l'indentation, etc... Bien que la technique soit simple, elle est relativement efficace. Effectivement, il y a un certain nombre de triches faites très rapidement la veille du rendu pour éviter d'avoir 0. Pour toutes ces triches, la méthode est efficace.

3.2 Reconnaissances des erreurs

Une remarque intéressante de [4] indique que deux programmes ayant réutiliser du code identique possède les même bogues et les même fonctionnalité. Nous n'avons trouvé aucun programme existant utilisant cette propriété

3.3 Ctcompare (Trichotomie)

Cette technique était utilisée par les Assistants 2005. Elle consiste à étudier l'apparition des mots clefs du langage étudié. Effectivement, les mots clefs représente l'algorithme utilisé (`for`, `if`, etc...). Deux fichiers ayant une suite de mots clefs identique utilisent sûrement le même algorithme. La méthode a porté ses fruits l'année dernière.

La figure 3.1 montre un exemple de sortie de se programme. Les Assistants 2005 ont amélioré ce programme en utilisant le programme `dotty`. `Dotty` permet de générer des graphes. On a ainsi pu créer des graphes symbolisant de les plus grandes parties de code communes entre les rendus. La figure 3.2 montre un exemple de graphe obtenu par cette méthode.

3.4 Simian

Simian est un logiciel commercial de Red Hill Consulting². Le principe est de transformer le fichier de manière à ne pas prendre en compte le nom des variables, le nom des fonctions, les littéraux, etc ... Par exemple, on change toutes les chaînes de caractères par "str", tous les noms de fonctions par "fun", etc ... Puis, on fait une comparaison de fichier ligne par ligne et on donne le plus grand nombre de lignes identiques consécutives. La méthode est proche de `ctcompare` mais, légèrement moins évoluée.

²<http://www.redhillconsulting.com.au/products/simian/>

3.5 CodeMatch

CodeMatch est un logiciel commercial³. Ce logiciel fait plusieurs tests :

- Comparaison ligne à ligne des fichiers
- Comparaison des commentaires
- Comparaison de la structure des algorithmes à l'aide d'une méthode proche de `ctcompare`.
- Recherche les identifiants identiques ou presque (par exemple Index et NewIndex) entre les fichiers.

Rien d'exceptionnel dans ce logiciel.

3.6 MyDropBox

MyDropBox⁴ est un projet commercial dont nous disposons peu d'information. La documentation indique que le projet utilise des "techniques d'IA".

3.7 Cogger[5]

Cogger utilise un CBR pour reconnaître les cas de triche. Les résultats de ce programme sont loin d'être satisfaisant.

La méthode utilisée par Cogger est décrite dans [5].

3.8 Sherlock

Sherlock est un projet Open Source. Il calcule tout d'abord une fonction de hash sur chaque token pour réduire le temps de comparaison par la suite (fichiers moins gros, donc, comparaison plus rapide). Sherlock fait ensuite une comparaison entre les fichiers et donne le nombre de mots en communs, quelque soit l'ordre. L'intérêt de la méthode et qu'elle est très rapide. On garde en mémoire une table de hash ayant pour clés les mots et pour valeur les fichiers contenant les mots en question. La méthode semble peu efficace, mais, elle est rapide et peut être utilisé pour élayer le domaine.

3.9 Moss[14]

Moss est un projet Open Source dont les techniques sont expliquée dans [14]. Le principe général est de repérer les mots inutiles dans les fichiers source, puis de les retirer (on considère ça comme du bruit) et de rechercher la plus longue sous séquence commune entre les deux fichiers.

3.10 Yap3[19]

Yap 1, 2 et 3 est un projet Open Source. Il se base sur les mêmes techniques que Moss et semble les pousser un peu plus loin. Tout d'abord, comme Moss, on retire les mots n'apportant pas

³environ 4000\$ pour une licence.

⁴<http://www.mydropbox.com>

d'informations des fichiers. Ensuite, Yap possède une liste de synonymes désignant des fonctions ou des mots clef proches⁵. Ensuite, on recherche la plus longue sous séquence commune.

3.11 Outils externes utiles

3.11.1 GNU C Compiler (`gcc`)

`Gcc` est le compilateur C de référence à EPITA. Il permet de transformer les fichiers sources en fichiers objets. Les techniques et les différentes représentations employées en interne peuvent nous être utiles à comprendre le code de l'élève.

3.11.2 Object Dumper (`objdump`)

Le rôle d'`objdump` est d'extraire un maximum d'informations d'un fichier objet. Il permet entre autre de désassembler un fichier objet.

3.11.3 `strip`

`strip` permet lui aussi de travailler sur les fichiers objets. Il permet de retirer du fichier objets toutes les données de manière à ne laisser que la partie exécutable de la binaire.

3.11.4 `strace`

`strace` est un utilitaire se chargeant grâce à `$LD_PRELOAD` permettant de récupérer les fonctions de la *libc* appelée par le programme. L'étude des appels de fonctions extérieure pourrait être intéressante. Effectivement, deux algorithmes identiques devrait avoir une liste d'appels de fonctions externes très proche. Néanmoins, cette méthode ne fonctionne que sur des programme faisant beaucoup d'appels à la *libc*. Or la plupart des algorithmes "intéressant" utilisent peu de fonctions de la *libc*⁶. De plus cette méthode implique l'exécution du programme et le passage dans l'intégralité du code pour avoir une chance de trouver la partie recopiée.

3.11.5 GNU Profiler (`gprof`)

`Gprof` est un utilitaire permettant de créer des *profils* d'un programme. Un profil indique entre autre le nombre d'appels de chaque fonctions et combien de temps le programme à passé dans chaque fonction. Tout comme avec `strace`, si deux programmes sont proches, leur profils devraient l'être aussi. Cette méthode est plus généralisable que celle de `strace`, mais, on doit, là encore, exécuter le programme avec un cas exécutant tout le code pour avoir une chance de trouver des points communs.

La figure 3.3 montre un exemple de sortie de `gprof`

⁵Par exemple : `strcmp/strncmp`, `for/while` ou bien `$/i/$j`, etc...

⁶On travaille généralement, sur des tableaux, des listes, etc...

FIG. 3.3 – Exemple de sortie de gprof

```
$ ruby -e 'puts "304*234675/4232+" * 30000 + "3"' | ./evalexp;
505710003
```

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
50.00	0.02	0.02	90001	0.00	0.00	stdin_to_int
25.00	0.03	0.01	480002	0.00	0.00	my_read
25.00	0.04	0.01	1	10.00	40.00	fn_parser
0.00	0.04	0.00	179999	0.00	0.00	fn_char_to_prior
0.00	0.04	0.00	179999	0.00	0.00	fop_char_to_op
0.00	0.04	0.00	90001	0.00	0.00	fn_next_num
0.00	0.04	0.00	30000	0.00	0.00	divide
0.00	0.04	0.00	30000	0.00	0.00	plus
0.00	0.04	0.00	30000	0.00	0.00	times
0.00	0.04	0.00	1	0.00	0.00	fn_len_base
0.00	0.04	0.00	1	0.00	0.00	fn_printnbr
0.00	0.04	0.00	1	0.00	0.00	fn_translate_base
0.00	0.04	0.00	1	0.00	0.00	my_printnbr
0.00	0.04	0.00	1	0.00	0.00	p_complete_params
0.00	0.04	0.00	1	0.00	0.00	p_treat_arg

3.11.6 `ctags` et `etags`

`Ctags` est un programme tout d'abord destiné à aider les éditeurs de texte à faire de la complétion sur les fonctions d'un code source. Il examine le code et en extrait les noms des fonctions, les endroits où elles se trouvent, etc... Ce genre d'informations pourraient nous être utiles pour repérer les noms en commun entre deux rendu.

3.11.7 `indent`

`indent` permet de réindenter du code. Il pourrait nous permettre de supprimer toutes les différences de style de code entre les élèves. Cette commande est très peu utile à EPITA vu que tous les étudiants doivent suivre la Coding Style [13] et que celle-ci laisse très peu de place pour le style personnel.

3.11.8 `wdiff` et `diff`

`Wdiff` et `diff` permettent de mettre en évidence les différences entre deux fichiers. `Wdiff` recherche les mots ajoutés et supprimés alors que `diff` travaille sur les lignes ajoutées et supprimées. Nous verrons dans la section 4.2 le fonctionnement de ces programmes et la manière de les adapter à nos besoins.

Chapitre 4

Solutions retenues

Sommaire

4.1 Empreinte de fichier	15
4.1.1 Trace d'exécution de programme	15
4.2 Distance entre les fichiers	15
4.2.1 Distance sur les mots rares ¹	16
4.2.2 La distance binaire	16
4.2.3 La plus longue sous-chaîne (Longest Common Substring, LCS)	16
4.2.4 Distance de Levenshtein	16
4.2.5 Greedy String Tiling (GST)	17
4.3 Optimisations	19
4.3.1 Clés MD5	19
4.3.2 Programmation dynamique	19
4.3.3 Karp-Rabin	20
4.3.4 Greedy String Tiling avec Karp-Rabin	20
4.3.5 Compression de tokens	22
4.3.6 Utilisation du cache	22
4.4 Fichiers à comparer	22
4.4.1 Analyse syntaxique	25
4.4.2 Analyse sémantique	26
4.4.3 Division en fonction	29
4.5 Conclusion	29
4.6 Les techniques "sioux"	29
4.6.1 Recherche d'incohérences	30
4.6.2 Recherche des relations sociales	30
4.6.3 Surveillance des comptes	30
4.6.4 Surveillance de la commande scp	30
4.6.5 L'obscurantisme	31
4.6.6 La psychologie	31
4.7 Expression du résultat et Techniques d'interprétation	31

¹Merci à Akim Demaille pour cette idée

4.7.1	Datamining	31
4.7.2	Système à base d'apprentissage unique (type Réseaux de Neurones) . . .	31
4.7.3	CBR	32
4.7.4	Système expert	32

PARMI les solutions proposées par les programmes existants, nous avons retenue certaines idées. Nous en avons améliorées et nous en avons trouvé d'autres.

4.1 Empreinte de fichier

Cette solution est tirée de Cogger [5]. Cette solution est aussi envisagée par Grier[8] en 1981 et Faidhi-Robinson [6] en 1987. A partir d'un fichier, on peut calculer un certain nombre de valeurs permettant de mettre en évidence certaines propriétés du fichier. Le principe est de calculer cette empreinte puis de repérer les fichiers ayant des empreintes proches dans la promo.

Nous utilisons les attributs suivants :

- Le nombre d'apparitions dans le fichiers de chaque mot clef du langage (`for`, `==`, etc ...).
- Le nombre total d'opérateurs.
- Le nombre d'identifiants uniques.

Nous pourrions en utiliser bien d'autres mais, nous sommes limités par le Coding Style d'EPITA [13].

4.1.1 Trace d'exécution de programme

Une remarque intéressante de [4] indique que deux programmes ayant réutilisé du code identique possède les même bogues et les même fonctionnalités. On peut ajouter qu'ils auront sûrement des fonctionnements identiques. On pourrait donc utiliser une empreinte basée sur le profile d'exécution du programme. Or `gprof` permet exactement de voir le profil d'exécution d'un programme. Nous ajoutons donc ces données à l'empreinte du programme.

4.2 Distance entre les fichiers

Alors que les empreintes ne se calculent que sur un fichier, on peut définir des pseudo-distances pour un couple de fichiers. Plus une pseudo-distance est faible plus les fichiers sont proches.

Nous allons utiliser plusieurs pseudo distances :

- La distance sur les mots rares
- La distance binaire
- La plus longue sous-chaîne commune
- La distance de Levenshtein
- La distance de Greedy String Tiling (GST)

4.2.1 Distance sur les mots rares²

L'idée vient du principe que si on retrouve une faute d'orthographe ou un mot ne faisant pas parti du vocabulaire courant dans deux fichiers, il y a des grandes chances que l'auteur soit le même [4].

On veut donc repérer les mots peu communs des fichiers et vérifier si ils n'apparaissent pas dans d'autres fichiers. Pour chaque tarball, on crée un tableau de tous les mots apparaissant³. Dès que l'on retrouve le mot dans n tarballs, on le supprime des listes et on le marque comme *courant*. Après avoir vérifier toutes les tarballs, il nous reste pour chaque tarball une liste de mots peu courants. La distance entre deux tarballs est l'inverse du nombre de mots peu courants qu'ils ont en commun.

4.2.2 La distance binaire

La distance binaire consiste à faire une comparaison caractère par caractère entre les fichiers. A partir du moment ou un caractère diffère, la distance est de 1, sinon, la distance est de 0.

Cette distance donne peu d'informations si la distance vaut 1 mais, si elle vaut 0, il y a des très forte probabilités que ce soit le même auteur⁴. De plus cette distance est très rapide et très optimisable.

4.2.3 La plus longue sous-chaîne (Longest Common Substring, LCS)

Le but est de trouver la plus grande partie commune entre les deux textes. Cet algorithme est très connu. Vous pourrez trouver les détails d'implémentation de cette algorithme dans [9] et [16]. Les commandes `diff` et `wdiff` utilisent des algorithmes proches de LCS. Nous ne nous attarderons pas sur cet algorithme car nous ne l'utiliserons pas.

4.2.4 Distance de Levenshtein

Le principe de cette distance est de trouver le minimum de correction à faire pour passer du premier fichier au second [11]. On mesure ensuite le pourcentage de texte inchangé.

Cette méthode est proche de celle utilisé dans les correcteurs orthographiques.

On choisit comme ensemble de corrections : ajouter un caractère et supprimer un caractère. On pourrait ajouter : échanger deux caractères et remplacer un caractère par un autre, mais, ces transformations offrent peu d'intérêt dans notre cas. On remarque l'on peut travailler sur des caractères (comme dans les correcteur orthographique), sur des mots (comme dans `wdiff`) ou sur des lignes (comme dans `diff`).

L'implémentation naïve (4.1) consiste à tester toutes les combinaisons possibles de corrections. On assiste alors à une explosion combinatoire.

La complexité de cet algorithme est alors de $O(3^n)$

Le principal problème de cet distance est qu'elle n'est pas capable de détecter le déplacement de texte.

²Merci à Akim Demaille pour cette idée

³On ajoute le numéro de la ligne où ils apparaissent pour que l'on puisse facilement les retrouver par la suite

⁴Il est possible que les deux personnes aient écrit le même texte

FIG. 4.1 – Implémentation naïve de la distance de Levenshtein

```

1  /*
   ** T1 and T2 are your texts
   ** t1 and t2 are pointer on texts
   */
   int correct(string T1, string T2, integer t1, integer t2)
   if (T1[t1] = end of T1 or T2[t2] = end of T2)
       return 0;
   ret = min(correct(T1, T2, t1 + 1, t2 + 1),
             correct(T1, T2, t1 + 1, t2),
             correct(T1, T2, t1, t2 + 1))
11  if (T1[t1] ≠ T2[t2])
       return ret + 1;
   return ret;

```

4.2.5 Greedy String Tiling (GST)

Jusqu'à présent, aucune distance ne permet de reconnaître des morceaux de textes échangés. GST consiste à associer deux à deux chaque lettres en essayant de les regrouper de manière à former les plus grandes suites communes possibles. La distance de GST est le nombre de caractères non associés dans la plus petite chaîne (le pattern).

Exemples (GST considère toujours la plus petites chaîne comme le pattern) :

Chaîne : abdca

Pattern : caab

On associe ca avec ca, puis ab avec ab. On compte le nombre de caractères restant dans le pattern. Nous obtenons une distance de 0.

Chaîne : abdca

Pattern : caaab

On associe ca avec ca, puis ab avec ab. On ne peut plus associé de caractères supplémentaire. On compte le nombre de caractère restant dans le pattern. Nous obtenons une distance de 1.

L'implémentation naïve consiste à rechercher la plus grande sous-chaîne commune puis, de rechercher les plus grandes sous-chaînes communes parmi les sous-chaînes restantes jusqu'à ce qu'il n'y ait plus de sous-chaînes communes. L'algorithme 4.2 formalise ce principe.

On remarque tout de suite que le principal problème de cette distance est son coût de $O(n^3)$. Nous verrons dans la partie suivante comment optimiser cet algorithme grâce à Karp-Rabin.

FIG. 4.2 – Implémentation naïve de la distance de GST

```

/*
** P is our pattern
** T is our Text
*/
function GST(string P, string T)
  lenghtOfTokenMarked = 0;
7   repeat
    /* Set minimum match lenght */
    maxMatch = minimumMatchLenght
    for each p ∈ unmarked tokens of P
      for each t ∈ unmarked tokens of T
        j := 0
        while (Pp+j = Tt+j and unmarked(Pp+j) and unmarked(Tt+j))
          j := j + 1
        if (j = maxMatch)
          add match(p, t, j) to list of matches
17      else if (j > maxMatch)
        create new list of matches
        add match(p, t, j) to list of matches
        maxMatch = j
        for m ∈ match(p, t, maxMatch)
          /* No substring occult m */
          if (m is not occulted)
            for (j := 0 to maxmatch - 1)
              markToken(Pp+j)
              markToken(Tt+j)
27      lenghtOfTokenMarked := lenghtOfTokenMarked + maxMatch
  until maxMatch = minimumMatchLenght

```

FIG. 4.3 – Optimisation de la distance de Levenshtein par programmation dynamique

```

/*
2  ** T1 and T2 are your texts
   ** t1 and t2 are pointer on texts
   ** tab is an array which save our results
*/
int correct(string T1, string T2, integer t1, integer t2)
if (T1[t1] = end of T1 or T2[t2] = end of T2)
    tab[t1, t2] = 0;
if (tab[t1 + 1, t2 + 1] does not exist)
    correct(T1, T2, t1 + 1, t2 + 1)
if (tab[t1, t2 + 1] does not exist)
12  correct(T1, T2, t1, t2 + 1)
if (tab[t1 + 1, t2] does not exist)
    correct(T1, T2, t1 + 1, t2)
ret = min(tab[t1 + 1, t2 + 1],
          tab[t1 + 1, t2],
          tab[t1, t2 + 1])
if (T1[t1] ≠ T2[t2])
    ret := ret + 1;
tab[t1, t2] = ret

```

4.3 Optimisations

4.3.1 Clés MD5

La distance binaire présentée en 4.2.2 peut facilement être optimisée dans notre cas. Le principe de cette optimisation est de calculer la clé MD5 de tous les fichiers dans un premier temps, puis de faire une comparaison sur les clés MD5 plutôt que sur les fichiers en eux même. Ceci permet de manipuler beaucoup moins de données.

4.3.2 Programmation dynamique

La distance de Levenshtein peut être considérablement optimisé en utilisant la programmation dynamique. La programmation dynamique peut-être utilisée si la solution optimale est une combinaison de sous-fonctions optimales. Dans ce cas, il existe un certains nombres de valeurs calculées plusieurs fois. Il suffit alors d'utiliser un tableau permettant de sauvegarder ces résultats. Il s'agit bien du cas qui se produit ici.

On obtient l'algorithme 4.3.

La complexité de l'algorithme est considérablement réduite. elle est maintenant de l'ordre de $O(n)$.

FIG. 4.4 – Implémentation naïve d’une sous-chaîne dans une chaîne

```

for  $i \in T$ 
  /* Compare strings */
  if (str_equals( $P$ ,  $T + i$ ))
    return OK
return KO

```

FIG. 4.5 – Optimisation à l’aide d’une fonction de hash

```

hashP = hash( $P$ );
for  $i \in T$ 
  hashT = hash( $T + i$ );
  if hashP = hashT
5   return OK
return KO

```

4.3.3 Karp-Rabin

Karp-Rabin est au départ utilisé pour la recherche une chaîne dans une sous-chaîne. Nous verrons plus tard qu’il sert de base à de nombreux algorithmes en dérivant (RKR-GST, etc ...).

Imaginons un pattern P à rechercher dans un texte T . L’implémentation naïve consiste comparer la première lettre de P avec la première lettre de T , si elle sont égale, on vérifie la seconde lettre, etc ... Si elles sont différentes, on teste à partir de la seconde lettre de T et ainsi de suite ... La figure 4.4 exprime ce principe.

Imaginons que l’on calcule une fonction de hash sur P et pour chaque sous-chaîne de T . On remarque alors que plutôt que d’utiliser la fonction `str_equals` pour faire la comparaison, il suffit de faire une simple comparaison d’entier. Le figure 4.5 illustre ce principe.

Le problème est que l’on doit calculer la fonction de hash taille de T fois. Karp-Rabin résout se problème en utilisant une fonction de de hash calculable en fonction de la valeur précédente, de l’ancien caractère et du nouveau :

$$\text{hash}(a, b, h) = ((h - a * 2^{m-1}) * 2 + b) \bmod q \quad (4.1)$$

Où m est le nombre de caractères sur lesquels on calcule la fonction de hash, a est le caractère qui ne doit plus rentrer en compte dans le résultat et b le nouveau caractère à prendre en compte. Enfin q est un nombre très grand. Dans la pratique, on utilisera la valeur maximum d’un entier pour q et m sera choisi de manière à être suffisamment important pour éviter les conflits, mais pas trop pour éviter les dépassements d’entier (31 est généralement une bonne valeur car elle permet de prendre en compte la retenue).

L’algorithme 4.6 décrit en détail le fonctionnement de l’algorithme de Karp-Rabin.

4.3.4 Greedy String Tiling avec Karp-Rabin

Tout d’abord, on remarque que l’on peut facilement améliorer l’algorithme de GST en appliquant des règles de programmation relativement simples :

FIG. 4.6 – Algorithme de Karp-Rabin (en C)

```
void KR(char *x, int m, char *y) {
    int d, hx, hy, i, j;

4   /* Preprocessing */
   /* computes  $d = 2^{m-1}$  with
      the left-shift operator */
   for (d = i = 1; i < m; ++i)
       d = (d << 1);

   for (hy = hx = i = 0; i < m; ++i) {
       hx = ((hx << 1) + x[i]);
       hy = ((hy << 1) + y[i]);
   }

14  /* Searching */
    j = 0;
    while (j <= strlen(y)-strlen(x)) {
        if (hx == hy && memcmp(x, y + j, strlen(x)) == 0)
            x + j is a solution;
        hy = (((hy - y[j]*d) << 1) + y[j + m])
            ++j;
    }

24 }
```

- On remarque qu’une fois les caractères marqués, ils ne sont plus utiles. On utilise donc une liste chaînée sur les tokens. Dès qu’un token est marqué, on le supprime de la liste.
- Pour chaque sous-chaîne non-marquée du Texte, on index chaque token.
- Si la distance entre deux caractères marqués est inférieure à `minimumMatchLenght`, on peut supprimer les caractères en question.
- Il faut mieux commencer pour les valeurs les plus grandes de `matchMax`. Il ne faut pas non plus commencer par des valeurs trop grandes, sinon, on fait des tours de boucles inutiles. Une bonne valeur de départ pour `MaxMatchLenght` est 20. Si par hasard on trouvait une chaîne commune de cette taille, on augmente cette valeur.
- Les sous-chaînes communes les plus grandes sont plus rare que les courtes. Par conséquent, lorsque l’on trouve un caractère en commun entre T_t et P_p , on comparera directement les caractères $T_t + \text{MaxMatchLenght}$ et $P_p + \text{maxMatchLenght}$ et on commencera à comparer à partir de là.

On obtient l’algorithme [4.7](#)

Ensuite, on remarque que le principe de fonctionnement de Karp-Rabbin peut être étendu aux algorithmes de recherches dans des chaînes présentés précédemment.

4.3.5 Compression de tokens

Jusqu’à présent, nous nous sommes limités à faire des distances sur des chaînes de caractères. Dans bien des cas, on voudra comparer par rapport à des mots voire à des lignes. Dans de tels cas, on considérera chaque mot/ligne comme une entité non sécable. La comparaison des mots/phrases pourra alors être grandement améliorée en compressant chaque entité. La compression permettra d’avoir moins de données à traiter et donc on utilisera moins de temps. A moins que le nombre d’entités différentes dépasse la capacité d’un int (c’est à dire plus de 4 milliards d’entités différentes), on peut se contenter de numéroter les entités dans l’ordre de leur apparition pour obtenir une compression suffisante.

4.3.6 Utilisation du cache

Cette dernière optimisation, est plus une remarque de programmation. Il y a parmi les algorithmes précédents, un certain nombre de valeurs devant être calculés pour chaque fichier. Il est évident que ces valeurs ne doivent être calculés qu’une fois et sauvegardés entre chaque utilisation.

4.4 Fichiers à comparer

Les distances présentées précédemment donnent de très bon résultats mais, elles ne prennent pas en compte la sémantique du texte. Effectivement, Les deux fonctions présentées en [4.9](#) sont absolument identiques :

Nous algorithmes (en particulier Levenstein et GST) trouverons bien une ressemblance très particulière sur ces fonctions. Néanmoins, on aimerait que nos algorithmes répondent une distance de 0 sur ces deux fonctions.

L’exemple [4.9](#) était simple. La figure [4.10](#) représente une fonction mieux maquillée.

Dans ce cas, nos algorithmes retourneront une distance assez forte alors que ces deux fonctions ont un comportement absolument identique.

FIG. 4.7 – Optimisation simple sur la distance de GST (en C)

```

GST(string P, string T)
lengthOfTokenMarked = 0;
repeat
  for each  $p \in$  tokens of  $P$ 
    while minimum_lenght( $p$ )
6       $p := p + 1$ 
      /* Set minimum match lenght */
      maxMatch = minimumMatchLenght
      for each  $t \in$  indexT[ $P_p$ ]
         $j :=$  maxMatch
        while  $P_{p+j} = T_{t+j}$  && is_contiguous( $P_{p+j}$ ) && is_contiguous( $T_{t+j}$ )
           $j := j - 1$ 
          if  $j = 0$  {
             $j :=$  maxMatch
            while  $P_{p+j} = T_{t+j}$  && is_contiguous( $P_{p+j}$ ) && is_contiguous( $T_{t+j}$ )
16               $j = j + 1$ 
              if ( $j =$  maxMatch)
                add match( $p, t, j$ ) to list of matches
              else if ( $j >$  maxMatch)
                create new list of matches
                add match( $p, t, j$ ) to list of matches
                maxMatch =  $j$ 
            }
          for  $m \in$  match( $p, t, j$ )
            /* No substring occult  $m$  */
26          if  $m$  is not occulted
            mk_not_contiguous( $T_{t-1}$ )
            mk_not_contiguous( $P_{p-1}$ )
            update_minimum_lenght( $P_{p-1}$ )
            update_minimum_lenght2( $P_{p+maxmatch+1}$ )
            for  $j \in [0 .. maxmatch - 1]$ 
              delete_from_index( $T_{t+j}$ )
            unlink( $P_p, P_{p+j}$ )
            unlink( $T_t, T_{t+j}$ )

36          lengthOfTokenMarked += maxMatch
until maxMatch = minimumMatchLenght

```

FIG. 4.8 – Suite de 4.7

```

update_minimum_length(P, p)
{
3  /* Mark end of last sub-string as small string */
   for i = 0; i < minimumMatchLength && is_contiguous(Pp-i); i++
       set_minimum_length(Pp-i)
   /* Delete it if sub-string is too small */
   if (is_uncontiguous(Pp-i))
       unlink(Pp-i, Pp)
}

update_minimum_length2(P, p)
{
13 /* Mark end of last sub-string as small string */
   for i = 0; i < minimumMatchLength && is_contiguous(Pp+i); i++
       ;
   /* Delete it if sub-string is too small */
   if (is_uncontiguous(Pp+i))
       unlink(Pp, Pp+i)
}

```

FIG. 4.9 – Exemple de deux fonctions sémantiquement identiques dont les identifiants ont été maquillés

```

1 int maxofthree(int x, int y, int z)
  {
    if ((x > y) && (x > z)) return (x);
    if (y > z) return (y);
    return (z);
  }

int bigtriple(int b, int a, int c)
  {
    if ((a > b) && (a > c)) return (a);
11    if (b > c) return (b);
    return (c);
  }

```

FIG. 4.10 – Exemple de deux fonction sémantiquement identique où le code à été changé

```

size_t      my_strlen(const char *str)
{
    size_t   length = 0;

    while (str != NULL && str[length] != '\0')
        length++;
7   return length;
}

int         my_strlen(char *s)
{
    int      i;

    for (i = 0; s && s[i]; ++i)
        ;
17  return i;
}

```

Jusqu'à présent, nous n'avons utilisé aucune propriétés du langage étudié. Pourtant, une analyse syntaxique et sémantique du du code pourrait nous être utile.

4.4.1 Analyse syntaxique

Tokenisation

La première méthode qui vient à l'esprit est de filtrer notre code source de manière à effacer les différences inutiles entre les codes. Pour cela, on fera (entre autres) :

- Ré-indentation du code
- Suppression des commentaires
- Renommage des identifiants

La figure 4.11 montre un exemple de tokenisation fait par `ctcompare`.

Réutilisation de l'AST du compilateur

On remarque que plus notre fonction de filtrage est performante et permet de comprendre le code, plus elle nous sera utile. Si on pousse le raisonnement jusqu'au bout, on remarque que l'on voudrait parser l'intégralité du code. Or le compilateur fait déjà ce travail de manière parfaite. Il est inutile de refaire un code qui existe déjà, il suffit de demander à `gcc` l'arbre de représentation intermédiaire qu'il possède en interne.

La récupération de cette représentation nous permet de nous abstraire de toutes le maquillage syntaxique du code. Ainsi le premier exemple 4.9 sera plus facilement reconnu. Néanmoins, cette représentation ne nous permet pas de reconnaître le l'exemple 4.10. Nous devons faire une analyse sémantique pour voir que les fonctions sont identiques.

FIG. 4.11 – Exemple de tokenisation (Tokenisation de `ctcompare`)

Code d'origine :

```

void    print_word(char *str)
{
3  char  *c;

    c = str;
    while ((c != '\\0') && (c != ' '))
        putchar(c++);
    putchar('\\n');
}

```

Code tokenisé :

```

VOID IDENTIFIER 1 OPENPAREN CHAR MULT IDENTIFIER 2 CLOSEPAREN
LINE
OPENCURLY LINE
CHAR MULT IDENTIFIER 3 SEMICOLON LINE
LINE
IDENTIFIER 3 EQUALS IDENTIFIER 2 SEMICOLON LINE
WHILE OPENPAREN OPENPAREN IDENTIFIER 3 NOT EQUALS CHARCONST etc...

```

4.4.2 Analyse sémantique

Pour faire une analyse sémantique, ils nous faut absolument un compilateur. Par définition, un compilateur est obligé de comprendre le fonctionnement du code.

Utilisation des fichiers objets

Si on prend l'exemple 4.10, et que l'on compile les deux fonctions, on remarque que les codes objets obtenus sont absolument identiques. Effectivement, notre compilateur, va tout d'abord parser notre fichier de départ puis il va le modifier de manière à obtenir la fonction la plus optimisée. Vu que les fonctions ont exactement le même comportement, il y a des grandes chances que le compilateur crée le même code objet.

Néanmoins, nous devons respecter certaines règles pour que cette fonctionnalité soit utilisable.

- Tout d'abord, il faut compiler tous les fichiers sources avec la même version du compilateur et les mêmes options. Sinon, en fonction des options de compilation et du compilateur, les objets créés seront différents.
- Ensuite, il ne faut pas utiliser les options d'optimisation du compilateur. Lorsque le compilateur optimise, il change complètement le code de manière à ce qu'il soit le plus rapide. Il est possible que dans un code optimisé, certaines fonctions disparaissent ou que des lignes soit échangés, etc ... Nous ne voulons pas de ça.
- Enfin, il faut *striper*⁵ le code objet de manière à retirer tous les noms de fonctions et autres éléments inintéressants pour le fonctionnement de la fonction. On utilisera l'option `-s` de `strip` qui permet de retirer un maximum de données de la binaire (en particulier les noms

⁵Striper une binaire consiste à retirer tous les éléments inutiles de la binaire

FIG. 4.12 – Exemple de code sémantiquement identiques

Listing 4.1 – regalloc1.c

```

1  int    my_strlen(char *s)
   {
     int i;
     for (int i = 0; s && s[i]; ++i) ;
     return i;
   }

   int main(void)
   {
     int size = my_strlen("Hello World!");
11  printf("length: %dn", size);
     return 0;
   }

```

Listing 4.2 – regalloc2.c

```

int main(void)
{
  int size = 0;
  char *s = "Hello World!";
  for (size = 0; s && s[size]; ++size) ;
  printf("length: %dn", size);
7  return 0;
}

```

de fonctions exportées qui sont normalement contenues dans le fichier objet).

- L'utilisation de `-DNDEBUG` permet de retirer les conditions de la macro `assert`. C'est une précaution supplémentaire normalement inutile.

Voici donc la ligne utilisée :

```
gcc -DNDEBUG -c $< -o $@ && strip -s $@
```

La figure 4.12 montre deux codes sémantiquement identiques. On remarque qu'une fonction a été incluse dans l'autre. Ce changement a impliqué quelques modifications sur l'initialisation des variables. La figure 4.13 montre les résultats compilés de ces codes. On remarque que la fonction se retrouve facilement dans le code assembleur produit.

Utilisation du LIR

La méthode trouve facilement sa limite. Il suffit qu'une fonction soit insérée dans un autre ou que la fonction soit modifiée plus en profondeur (en redécoupant en sous-fonctions par exemple) pour que le compilateur n'alloue plus les registres de la même manière et que le code objet soit

FIG. 4.13 – Résultat compilé de codes sémantiquement identiques

Listing 4.3 – Résultat de la compilation de regalloc1.c

```

2      .file    "regalloc1.c"
      .text
globl my_strlen
      .type   my_strlen, @function
my_strlen:
      pushl  %ebp
      movl   %esp, %ebp
      subl  $4, %esp
      movl   $0, -4(%ebp)
L2:
      cmpl  $0, 8(%ebp)
12     je    .L3
      movl  -4(%ebp), %eax
      addl  8(%ebp), %eax
      cmpb  $0, (%eax)
      jne   .L4
      jmp   .L3
L4:
      leal  -4(%ebp), %eax
      incl  (%eax)
22     L3:
      movl  -4(%ebp), %eax
      leave
      ret
      .size  my_strlen, .-my_strlen
      .section .rodata
LC0:
      .string "Hello World!"
LC1:
      .string "length: %d\n"
32     .text
globl main
      .type   main, @function
main:
      pushl  %ebp
      movl   %esp, %ebp
      subl  $8, %esp
      andl  $-16, %esp
      movl  $0, %eax
      subl  %eax, %esp
42     subl  $12, %esp
      pushl $.LC0
      call  my_strlen
      addl  $16, %esp
      movl  %eax, -4(%ebp)
      subl  $8, %esp
      pushl -4(%ebp)
      pushl $.LC1
      call  printf
      addl  $16, %esp
52     movl  $0, %eax
      leave
      ret
      .size  main, .-main
      .section
.note.GNU-stack,"",@progbits
      .ident "GCC: (GNU) 3.3.2"

```

Listing 4.4 – Résultat de la compilation de regalloc2.c

```

      .file    "regalloc2.c"
      .section .rodata
LC0:
      .string "Helo World!"
LC1:
      .string "length: %d\n"
      .text
globl main
      .type   main, @function
main:
      pushl  %ebp
      movl   %esp, %ebp
13     subl  $8, %esp
      andl  $-16, %esp
      movl  $0, %eax
      subl  %eax, %esp
      movl  $0, -4(%ebp)
      movl  $.LC0, -8(%ebp)
      movl  $0, -4(%ebp)
L2:
      cmpl  $0, -8(%ebp)
      je    .L3
23     movl  -4(%ebp), %eax
      addl  -8(%ebp), %eax
      cmpb  $0, (%eax)
      jne   .L4
      jmp   .L3
L4:
      leal  -4(%ebp), %eax
      incl  (%eax)
      jmp   .L2
L3:
33     subl  $8, %esp
      pushl -4(%ebp)
      pushl $.LC1
      call  printf
      addl  $16, %esp
      movl  $0, %eax
      leave
      .size  main, .-main
      .section
.note.GNU-stack,"",@progbits
43     .ident "GCC: (GNU) 3.3.2"

```

Représentation	Mots rares en commun	Binaire	LCS	Levenshtein	GST
Code brute	++	--	-	-	-
Code tokenisé/AST ⁶	--	+	+	+	++
LIR ⁶	--	++	++	++	++
Assembleur ⁶	--	+	-	-	-
Assembleur réduit ⁶	--	+	-	-	-
Objet	--	+	-	-	-

Légende :

- Aucun ou pas applicable
- Peu d'intérêt
- + Un peu d'intérêt
- ++ Très intéressant.

^fToutes ces représentations sont divisées en fonction pour de meilleurs résultats

différent. Heureusement, l'allocation des registre est la dernière étape de compilation dans un compilateur. Il existe une représentation intermédiaire appelée LIR (Low Intermediate Representation) très proche de l'assembleur. Dans cette représentation, la plupart des optimisation de code ont déjà été faite, mais on considère que le nombre de registre est encore infini.

4.4.3 Division en fonction

On remarque que l'on à jusqu'à présent travailler sur des fichier entiers. Dans la plupart des représentations décrites ci-dessus, on peut facilement séparer les fonctions entre elles. Seul le fichier objet nous empêche de le faire. Il suffit alors de transformer le code objet en assembleur (le code objet et l'assembleur sont absolument bijectifs) pour que l'on puisse facilement le découper en fonction. Le travail sur des fonctions plutôt que sur des fichier nous permet d'avoir une granularité plus petite et donc d'avoir des meilleurs résultats.

4.5 Conclusion

Le tableau suivant résume les différentes distances présentées et les formes de sources sur lesquels elles sont appliquées. On a noté si la distance était applicable et si elle apportait un résultat intéressant à la détection de la triche.

Le LIR semble être la représentation la plus avantageuse, néanmoins, chaque paires représentation/distance à ces avantages et ces inconvénients. le mieux est donc de calculer les différentes paires distance/représentation intéressantes.

4.6 Les techniques "sioux"

Aux techniques présentées précédemment, nous ajoutons certaines remarques utiles dont nous avons l'idée.

4.6.1 Recherche d'incohérences

Le principe est de rechercher des incohérences dans le rendu de l'élève. Ces incohérences sont généralement des preuves faibles, mais, elles sont à prendre en compte et peuvent élever le doute sur un rendu.

La plupart de ces incohérences se trouvent au niveau de l'entête du fichier. A EPITA, l'entête sur les fichiers est obligatoire. Une personne ayant repris un fichier va essayer de maquiller l'entête pour faire croire qu'elle l'a fait. On peut facilement maquiller parfaitement le fichier de manière à ne laisser aucune incohérence. Cette technique permet principalement de repérer les personnes reprenant le fichier d'une personne dans la panique juste avant le rendu et qui oublie de maquiller parfaitement le fichier.

L'incohérence la plus importante est la présence d'un login différents de l'auteur dans le rendu.

On vérifie ensuite que le nom du fichier dans l'entête est le même que le fichier⁷

On vérifie que la date de création du fichiers et la date de dernière modification (dans le header et sur les fichiers) ne sont pas trop proche. Cela indique à coup sur que le fichier à été copier, puis l'entête recrée.

On recherche des fichiers ayant une date de modification antérieure au sujet.

4.6.2 Recherche des relations sociales

Le risque de triche est plus important entre des personnes se connaissant. Il arrive souvent que un élève "aide" un ami en lui donnant son projet juste avant le rendu. Le but de cette méthode serait de trouver les relations sociale à l'intérieur de la promotion et de surveiller plus particulièrement les personnes se connaissant. Pour créer ces ensembles de relations, nous pouvons regarder les groupes d'élèves travaillant ensembles dans les projets à faire par groupe.

Il n'est pas prévu d'implémenter cette technique. Elle est néanmoins très intéressante. On pourrait la pousser très loin.

4.6.3 Surveillance des comptes

Les tarballs à EPITA ont toutes un nom du type `login-projet.t*`. Login est le login du chef de projet dans les projet se faisant en groupe. Une des méthodes les plus efficaces consiste à surveiller l'apparition de fichier possédant des noms de tarballs ne correspondant pas au login du propriétaire du fichier sur les comptes des élèves. On recherche ensuite les tarballs dont le propriétaire est dans le même groupe de projet que le login du nom de la tarball. La liste obtenue est probablement une sources d'inspiration pour leurs propriétaires.

Cette technique n'est pas utilisée par étique. Effectivement, il est techniquement possible d'aller espionner les comptes des élèves, mais, nous refusons de recourir à de tels procédés

4.6.4 Surveillance de la commande scp

La commande `scp` permet de copier un fichier d'un compte à un autre compte. Il suffit d'espionner cette commande pour repérer tous les passages de fichiers source⁸. Cette technique serait

⁷Cette incohérence peut être produite par le renommage de fichier par l'auteur réel

⁸On peut facilement repérer les fichiers source grâce à leur extensions : `.sh`, `.pl`, `.cc`, `.hh`, `.c`, `.h`, `.java`, `.tar`, `.gz`, `.bz2`, etc ...

très efficace. Néanmoins, tout comme la technique précédente, nous nous refusons de la mettre en pratique par étique. De plus, la contrainte technique est assez grosse et il nous faudrait la collaboration des administrateur du réseau d'EPITA. A cause de cette collaboration et de cette mise en place difficile, il est fort probable que la technique soit rapidement éventée.

4.6.5 L'obscurantisme

Un des points les plus importants des techniques de recherche de tricheurs est l'obscurantisme. Les élèves ne doivent surtout pas savoir quelles sont les techniques employées⁹. Toutes les techniques utilisées sont assez facilement contournables. Notre principal atout est que les élèves ne connaissent pas les techniques et qu'elles ne peuvent donc pas être facilement contournées.

4.6.6 La psychologie

L'autre point important allant de paire avec l'obscurantisme est psychologie des élèves. Il est important de montrer les cas de triches repérés de manière à ce que les élèves ne tentent pas de tricher.

4.7 Expression du résultat et Techniques d'interprétation

A l'aide de toutes les techniques précédentes, nous avons obtenu une série de mesures. Il nous faut maintenant interpréter ces mesures pour savoir si nous avons à faire à un cas de triche ou non.

4.7.1 Datamining

La première idée consiste à utiliser le Datamining pour classer chaque paire de rendus. Le but serait donc de faire apparaître une classe de tricheur. Cette méthode pose un problème. Effectivement, La topologie du problème n'est pas adapté. Le Datamining va trouver des classes, mais, les chances que l'une d'entre elle soit la classe "tricheurs" sont faibles. Il n'y a pas de raisons que le Datamining divise l'ensemble selon les classes que l'on souhaite. Il faudrait pour résoudre le problème trouver une autre représentation des cas (changer de base). Mais, si l'on arrivait à trouver cette représentation, on aurait quasiment résolu notre problème et le Datamining deviendrait inutile.

4.7.2 Système à base d'apprentissage unique (type Réseaux de Neurones)

Nous pourrions utiliser un réseau de neurones pour classer nos cas. Cette technique pose deux problèmes :

- Le système ne reconnaîtra que les cas de triches qu'il a déjà rencontré pendant l'apprentissage. De plus, l'ajout d'un nouveau type de triche impose de refaire tout l'apprentissage.
- Nous n'avons pas de base d'exemples à faire apprendre à notre système. Nous avons effectivement tous les anciens rendus, mais, nous sommes incapable de savoir si la personne était un tricheur ou pas.

⁹C'est la raison pour laquelle la publication de ce rapport doit être restreint

4.7.3 CBR

Le CBR est un système à base d'apprentissage qui fait exception aux règles précédentes car il s'agit d'apprentissage incrémental. On commence par ajouter les premiers exemples et dès que le CBR n'arrive pas à classer un exemple, il demande à l'utilisateur si c'est un tricheur ou non.

Néanmoins, pour que le CBR apprenne, il aura besoin qu'on le guide assez souvent au début. Sans oublier que si l'on ajoute des techniques supplémentaire, une bonne partie de l'apprentissage sera à refaire.

De plus, les assistants sont un groupes d'étudiants qui change tous les ans. Nous savons par expérience que les gros projets deviennent très rapidement non maintenables et difficiles à utiliser par manque de temps pour former le reste du groupe. De plus, il faut justifier tous les crédits supplémentaires dont nous avons besoin. Il nous faut donc éviter les gros projets et préférer les petits projets agiles, facilement adaptables avec un code facilement appréhendable pour nos successeurs. Or, la mise en place d'un CBR ressemble à une "usine à gaz". Il y a beaucoup de chose à faire, il devient très vite difficilement maintenable, peu adaptable et peu utilisable si il est mal conçu. Ce serait sûrement une excellente solution, mais, elle ne serait jamais en place avant plusieurs semaines ¹⁰.

4.7.4 Système expert

Nous devons donc trouver une solution ayant des coûts (au sens large) de développement plus réduits. Le système expert semble être un bon compromis. La différence avec le CBR est que les utilisateurs devront entrer les règles à la main plutôt que le système les trouvent tout seul. Dans notre cas, ça n'est pas si gênant que ça. Les règles sont généralement simples. L'utilisateur sait quel cas il veut prendre en compte. Ce système sera immédiatement opérationnel et les règles se préciserons avec les années.

De plus, les utilisateurs sont des codeurs, ça ne pose donc pour eux aucun problème de créer des règles. Ainsi plutôt que de créer un moteur de systèmes experts, nous préférons l'englober dans un langage déjà existant. Ceci évitera à l'équipe d'apprendre un nouvel outil ¹¹ et évite des frais de développement.

¹⁰A condition de faire travailler quelqu'un à plein temps sur le projet, ce qui est utopique

¹¹rappel : pas de temps/budget pour les transferts de connaissances

Chapitre 5

Implémentation

Sommaire

5.1 Plateforme de développement : TestTKewl	33
5.1.1 Récapitulatif des besoins	33
5.1.2 Présentation de l'outil	34
5.1.3 Architecture du projet	36
5.1.4 Ecriture d'un test	38
5.1.5 Langage d'entrée	39
5.1.6 Le filtrage	40
5.1.7 Autres particularités	41
5.1.8 Chaîne d'utilisation	43
5.1.9 Bilan	44

Nous allons voir dans cette partie comment nous allons faire pour utiliser toutes les différentes solutions, vues dans les parties précédentes, ensemble. L'idée est d'utiliser une plateforme de test qui va coordonner l'ensemble. En effet nous avons pu nous apercevoir que nous disposons d'un large panel d'outils pour récupérer des informations sur le code source mais aucune n'est à 100% fiable. Nous allons donc vouloir faire collaborer tous ces outils afin qu'ensemble ils fournissent une information pertinente à l'utilisateur.

L'outil que nous avons choisit de développé n'est pas directement lié avec le cas de la triche, mais est plus généralement une plateforme de test. Nous allons dans cette partie d'abord décrire le fonctionnement de cet outils dans son utilisation normale, puis voir comment nous l'avons utilisé dans le cas particulier de la triche.

5.1 Plateforme de développement : TestTKewl

5.1.1 Récapitulatif des besoins

L'outil TestTKewl est la pour résoudre la problématique de test d'une entité, une problématique très courante dans le processus de validation (ou de qualité) d'un logiciel.

Il existait déjà des solutions générique essayant de répondre à cette problématique entre autre une développée par les assistants 2003, et maintenant maintenue par nous, mais aucuns de ces

outils ne répond bien aux besoins. Les spécifications complètes de l'outil idéal ont été rédigées et se trouvent dans l'annexe A de ce document.

Voyons plus en détail les différents besoins de cette problématique.

Langage de manipulation de fichiers

Les scénarios d'utilisation de TestTKewl est le suivant : on veut appliquer sur une entité en entrée une suite de traitements et avoir en sortie une trace du déroulement de ces différents traitements.

On peut voir cela comme appliquer une suite de fonctions à des arguments et en récupérer une trace de l'exécution en résultat. Cette opération est bien souvent facile quand l'on travaille sur des valeurs (nombre entier, structures) grâce à la puissance des langages (procéduraux comme fonctionnels), mais bien plus fastidieuse lorsqu'on veut l'appliquer à des fichiers (exécutables, tarballs, etc, ...). Ce que l'on aimerait donc c'est de disposer d'un langage permettant d'exprimer ce genre de scénario sur des fichiers simplement.

De plus le nombre de situations différentes et de cas particulier que l'on est amené à rencontrer d'un projet à un autre fait qu'on veut pouvoir exprimer n'importe quel traitement.

TestTKewl doit donc offrir un pouvoir d'expression égal à tout langage de programmation, mais être spécialisé dans son domaine afin de réduire le nombre de lignes à écrire.

Différentes étapes

Comme nous l'avons vu pour la triche il s'avère que nous avons souvent besoin de plusieurs phases de traitement, il faut une phase qui génère les données et ensuite une phase qui sélectionne l'information pertinente. Ces deux phases ne peuvent pas être résumées en une seule car bien souvent d'un contexte à un autre les données pertinentes ne sont pas les mêmes. Par exemple certaines fois nous voudrions savoir combien il y a de tests qui passent et d'autres fois uniquement le nom des tests. Pour cela il n'y a pas besoin de régénérer les données, sauf si on veut changer les conditions de test.

TestTKewl devra donc proposer plusieurs une phase de génération de l'information et une (ou plusieurs) phases de traitement de l'information.

Diversité des tests, et de la sortie

Tester un projet ne se résume pas en une action simple, c'est souvent composé d'un grand nombre d'action de type et de nature différentes. Par exemple on va d'abord vouloir voir si le projet compile, s'il s'installe bien, puis on va vérifier si l'exécution correspond bien aux spécifications, et ainsi de suite. Chacune de ces tâches est différente de l'autre mais elles doivent toutes pouvoir être imbriquées les unes avec les autres afin de pouvoir tester n'importe quel type de projet, on veut aussi pouvoir rajouter des tests facilement.

De plus la trace résultante doit pouvoir se conformer aux volontés de l'utilisateur et doit donc être facilement interchangeable.

TestTKewl doit donc être modulaire, que ce soit au niveau des tests que de la trace d'exécution finale.

5.1.2 Présentation de l’outil

Contexte et auteurs

Le projet TestTKewl est développé à l’extérieur du cadre des assistants car se veut d’une plus grande envergure. En effet il n’existe pour l’instant pas d’équivalent. Il est distribué sous licence GPL et devrait à terme apparaître dans tous les sites de logiciels libre tels que savanha.org ou sourceforge.org.

Le projet a été projeté, défini et initié par nous, notamment suite à la première expérience que nous avons eu avec la première “moulinette” de test. Nous ont rejoint, intéressé par un projet du même type, deux élèves du Laboratoire de recherche et développement d’EPITA : Nicolas Pouillard et Nicolas Despres. Ceux ci ont implémenté les plupart des idées que nous avons eu en ajoutant leur lot de nouveautés. De plus il a fallu intégrer TestTKewl à l’environnement des assistants déjà présents.

Principe

TestTKewl prend la forme d’un framework de test, c’est-à-dire un ensemble de tests de base permettant de résoudre la plupart des situations courantes. L’idée étant qu’à partir de ces tests on pourra facilement grâce à la surcharge de méthode et l’héritage redéfinir certain comportement afin qu’il s’adapte au projet courant. Ainsi l’écriture de test s’en voit fortement facilité et le temps réduit.

Au départ TestTKewl ne fournit que quelques stratégies de test et des classes abstraites. Par contre, TestTKewl s’agrandi par la contribution des utilisateurs qui publient leurs stratégies de tests.

En plus des différentes stratégies de tests il est fournit un programme coordinateur qui permet d’accompagner l’utilisateur dans toutes les phases de tests, de l’écriture des stratégies de tests à l’exploitation des résultats en passant par l’exécution des tests.

Exemple

Voici un exemple de fichier d’entrée décrivant une suite de tests à exécuter :

```
#exit.yml
---

name: Test the exit status verification (exit.yml)
class: TestTSuite
attributes:
  class: TestTCmd
content:
  - name: true
    command: true
    exit: 0
  - name: false
    command: false
    exit: 1
```

```
- name: false
  command: false
  exit: 42
```

Voici les différentes actions qui sont décrites dans ce fichier :

- le premier test nommé “true” exécutera la commande `true` et s’attend à avoir une valeur de retour de 0, si c’est le cas le test sera considéré comme bon sinon comme faux.
- le deuxième test nommé “false” exécutera la commande `false` et on s’attend à recevoir 1 comme valeur de retour.
- le troisième test ressemble au deuxième sauf que la valeur de retour attendu est 42.

Les premières lignes regroupent les trois tests en un seul test grâce à un test particulier qui s’appelle `TestTSuite`. Cette stratégie de test crée une arborescence dans nos tests et permet de structurer les tests.

Il est à noter que le fichier d’entrée est écrit dans un des langages possibles, le plus simple, mais le moins expressif. On verra par la suite en détails les langages d’entrée.

Les tests s’exécutent avec la commande suivante :

```
ttk exit.yml
```

Cette commande se charge de charger le fichier d’entrée contenant la description des tests à exécuter, de les exécuter et de reporter la trace d’exécution à l’écran.

Le résultat obtenu est le suivant :

```
class: TTK::Test::TestTSuite
content:
- name: Test the exit status verification (exit.yml)
  class: TTK::Test::TestTSuite
  attributes:
    class: TestTCmd
  content:
    - name: true
      command: "true"
      exit: 0
      output_status: PASS
      error_status: PASS
      status: PASS
    - name: false
      command: "false"
      exit: 1
      output_status: PASS
      error_status: PASS
      status: PASS
    - name: false
      command: "false"
      exit: 42
      output_status: PASS
      error_status: PASS
```

```
my_exit: 1
status: FAILED
```

Sur la trace apparaît le détail de l'exécution. Le TestTCmd lance une commande puis vérifié sa sortie standard et sa sortie d'erreur ainsi que son code de retour. Nous pouvons constater que pour les trois tests la sortie d'erreur et la sortie standard sont bon et qu'il n'y a rien à signaler, normal puisque nous n'avons mis aucune contraintes sur ces éléments et que les programmes `true` et `false` ne font aucun affichage.

Pour chaque test s'affiche l'affichage attendu (`exit`) et la sortie constatée (`my_exit`). On remarque que pour le dernier tests ces deux valeurs ne sont pas égales, le test est donc reporté comme faux.

Bien évidemment l'affichage obtenu n'est qu'un exemple, il est complètement paramétrable afin de répondre à tous les besoins.

5.1.3 Architecture du projet

Le projet s'articule autour de trois grands axes :

- le *loader*, il lit le langage d'entrée et crée les tests ;
- les *tests*, il représente un élément à vérifier ;
- les *filtres* et le *dumppers*, permettent d'afficher le resultat obtenu.

Comme nous pouvons le voir sur la figure 5.1, l'exécution du logicielle se décompose en plusieurs phases :

- le *loader* lit le fichier d'entrée et détermine la stratégie de test, il crée en mémoire la hiérarchie de tests contenant les bonnes valeurs.
- chaque test sait s'exécuter, on exécute alors le test racine se qui produit une réaction en chaîne.
- à chaque test est passé un afficheur en paramètre, le test va écrire dedans les différents attributs le caractérisant. Les données reçus sont filtrés par les filtres afin d'en retenir uniquement les informations importantes, ou faire un post traitement, et enfin elles sont affiché dans le format de sortie grâce au *dumper*.

5.1.4 Ecriture d'un test

L'écriture d'un test doit rester quelque chose de rare, et donc TestTKewl fournit de base un grand nombre de Tests aux utilisateurs. Ceux-ci sont fait de façon à facilement être réutilisés et résolvent les problématiques les plus communes.

La vie d'un test se décompose en trois parties :

prologue : elle est exécutée avant de créer un nouveau processus, il crée le test à l'aide des différentes valeurs qui ont été chargé depuis le fichier de configuration. Par exemple, pour un test qui compile un fichier, il va regarder l'extension du fichier, et à partir de cela créer la ligne de commande exacte à exécuter.

milogue : c'est le coeur du test, il est exécuté dans un processus à part.

epilogue : après l'exécution du test on récupère le code de retour, la sortie du test et d'autres informations utiles. Elles sont alors interprétées pour savoir si le test a réussi ou non. On peut aussi afficher des informations complémentaire sur l'exécution du test.

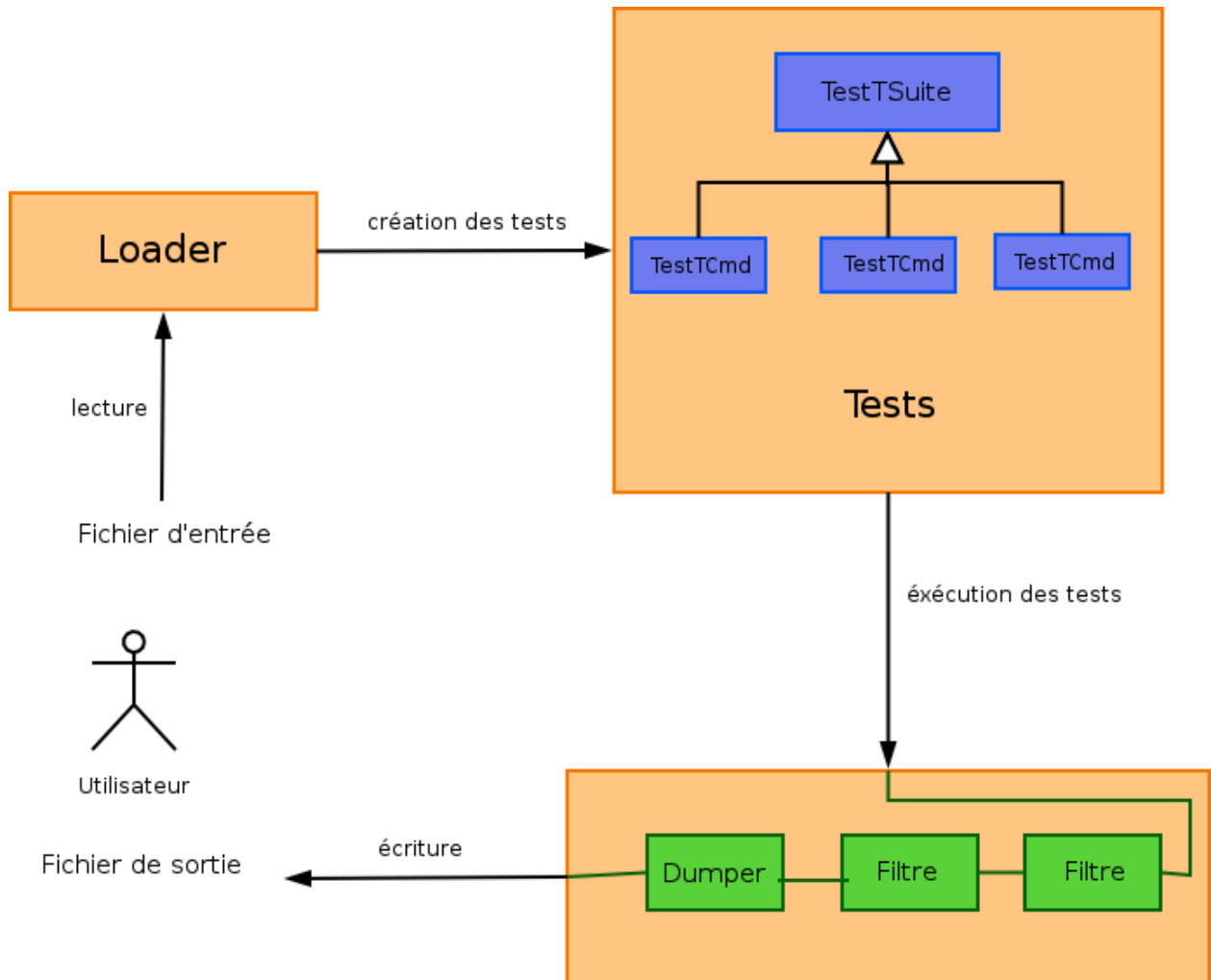


FIG. 5.1 – Détails des différentes étapes de TestTKewl

Il arrive fréquemment qu'un utilisateur veuille préciser le comportement d'une partie du test. Par exemple, au lieu de comparer la sortie standard à l'aide de la commande `diff`, on peut vouloir comparer en ignorant toutes les occurrences d'un mot, mais pourtant le test reste identique sur tous ces autres aspects. Cela est très facilement réalisable grâce à la modélisation objet. Il suffit d'hériter du test que l'on veut modifier et puis de surcharger la méthode qui ne nous satisfait pas. Bien souvent il suffit simplement d'écrire une méthode qui appelle `super` (la méthode mère) et qui ensuite fait le traitement particulier. Tous les tests sont écrits de façon à rendre cette démarche la plus simple possible.

Une des choses qui pourrait faire perdre beaucoup de temps, est toute la partie configuration du test. Pour cela chaque attribut va avoir plusieurs propriétés, à l'image des frames. On va pouvoir dire :

- le nom de l'attribut, par lequel il sera accessible de l'extérieur pour la configuration ;
- une description de l'attribut ;
- si l'attribut est obligatoire, c'est à dire que s'il n'est pas initialisé l'exécution du test produira une erreur (une valeur par défaut compte pour une initialisation) ;
- une valeur par défaut ;
- si la valeur de l'attribut doit être affichée dans le résultat.

Tout cela se fait à l'aide d'un simple appel à la méthode `attribute`.

En plus de tout ça on peut aussi indiquer :

- un bloc de commande à exécuter lorsqu'on accède à l'attribut ;
- un bloc de commande à exécuter lorsqu'on affecte une valeur à un attribut, cela sert notamment à convertir le type d'un attribut, par exemple sa valeur initiale est donnée sous forme de chaîne de caractère dans le fichier d'entrée et est convertit en entier lors de son initialisation.

Tout cela se fait en surchargeant les accesseurs des attributs, où en s'aidant de la méthode `method_missing` qui est la méthode par défaut qui est appelée quand un objet ne sait pas répondre à un certain message, il sert à gérer l'imprévu en quelques sortes. Cette conception peut se rapprocher de l'idée de démon des frames, et `method_missing` de `if_needed` en particulier.

5.1.5 Langage d'entrée

Les limites d'un fichier de configuration

L'entrée de notre programme est plus ou moins toute la configuration que nous voulons faire. Nous allons, par exemple vouloir configurer ce genre de choses :

- les différents tests que l'on va vouloir effectuer ;
- l'ordre de chacun de ces tests ;
- les valeurs de références de chaque tests ;
- où récupérer chaque tarball ;
- comment extraire le nom de l'auteur à partir d'une tarball ;
- le poids dans la note de chaque test ;
- ...

Cette liste est loin d'être exhaustive, chaque jours apparaissent de nouveaux cas d'utilisation.

Dans un tel cas on a vite fait de voir qu'on ne pourra jamais prévoir tous les besoins des utilisateurs et que notre fichier d'entrée devra dynamiquement s'adapter aux informations qu'on lui donne. C'est pour cela qu'on utilisera pas un fichier statique mais un langage dynamique.

TAB. 5.1 – Attributs de chaque tests créés

	Test 1	Test 2
name	I'm suppose to pass	I'm suppose to fail
command	true	false
exit	0	0

Un cas simple

Dans l'exemple en début de section nous avons vu que notre fichier d'entrée contient une liste de tests à effectuer avec leur paramètres. En interne nous allons pouvoir exprimer le même genre de chose ainsi :

```
aSuite = Test::TestTSuite.new
aSuite.name = 'A set of tests'
aSuite.loader = Loader::Yaml.new
t1 = aSuite.create(Test::TestTCmd)
t1.name = "I'm suppose to pass"
t1.command = "true"
t1.exit = 0
aSuite.create({ 'name'      => "I'm suppose to fail",
                'command'  => "false",
                'exit'     => 0
              })
```

On déclare ici un ensemble de test (*TestTSuite*) contenant deux tests qui sont créés de deux façons différentes. Le deux tests ont les attributs suivants :

5.1.6 Le filtrage

But et principe

Un des points très important dans le framework est la sortie pour l'utilisateur. Celle ci doit contenir toutes les informations utiles pour comprendre les tests qui ont été exécutés et leur résultat, mais souvent on veut avoir des informations précises et rapidement et non une masse d'informations.

Le principe est donc, par défaut on affiche tout, et ce sont des filtres qui vont permettre de sélectionner les parties qui nous intéressent, et pourquoi pas d'en extraire, voir de produire, de l'information.

Notre représentation interne du résultat est un arbre YAML, composé de tables de hachage et de tableaux, le tout forme donc un arbre n-aire. Le but d'un filtre est donc de transformer cet arbre vers un autre.

Dans la suite nous allons utiliser cet exemple donné sous la forme YAML :

```
name: Test the project
class: TestTSuite
tests:
```

```

- name: Test ex_00
  class: TestTCmd
  command: ./ex_00 toto
  output: OK
  error: OK
  code: OK
  status: PASS
- name: Test ex_01
  class: TestTCmd
  command: echo 'titi' | ./ex_01
  output: KO
  error: OK
  code: OK
  status: FAIL

```

La transformation se fait à base de règles. Les règles sont composées de deux éléments :

- Le chemin vers les éléments à traiter. Ce chemin est donné dans un langage propre qui s’inspire de XPath. Par exemple : `’ /class=TestTSuite.* /class=TestTCmd’` va chercher toutes les tests qui sont des TestTCmd (`class=TestTCmd`) contenu dans (la balise `’ / ’`) une TestTSuite ou une classe dérivée (`class=TestTSuite.*`) en partant de la racine de l’arbre (le `’ / ’` initial). On obtient alors dans cet exemple les deux tests de la TestTSuite.
- Un bloc de commande a exécuté sur chaque test. Ce bloc de commande va permettre de sélectionner ou d’interpréter l’élément et de construire un nouvel arbre contenant le résultat du filtre. On pourra par exemple afficher le nom du test et son statut pour avoir un récapitulatif de tous les tests rapidement et facilement.

Amélioration : compilation de règles

Ce principe est très similaire à celui de XSTL, un outil de transformation très utilisé pour le XML. Mais l’un des majeurs défaut du XSLT est qu’il est très lent, en effet pour chaque règle (un chemin et un bloc), il va parcourir tout l’arbre pour trouver les éléments qui répondent au chemin, puis va appliquer le bloc sur tous les éléments.

Pour améliorer ce principe nous allons reprendre l’idée de l’algorithme RETE, c’est à dire compiler les règles. On fait donc une première phase où toutes les règles sont déclarés et compilés dans un graphe. Le graphe contient trois types de noeuds :

Node : représente une condition de passage pour passer d’un étage de l’arbre à un autre, composé du nom d’une balise (la valeur de gauche) et d’une liste de Value ;

Value : indique la condition à respecter pour que le chemin soit valide et pointe vers la suite du chemin ;

Match : composé d’un bloc de commande à exécuter. Indique la fin d’un chemin.

Par exemple si on a les règles suivantes :

```

/class=TestTSuite.* /class=TestTCmd => print_cmd
/class=TestTSuite.* /output=KO => check_output
status => count_all
status=PASS => count_pass

```

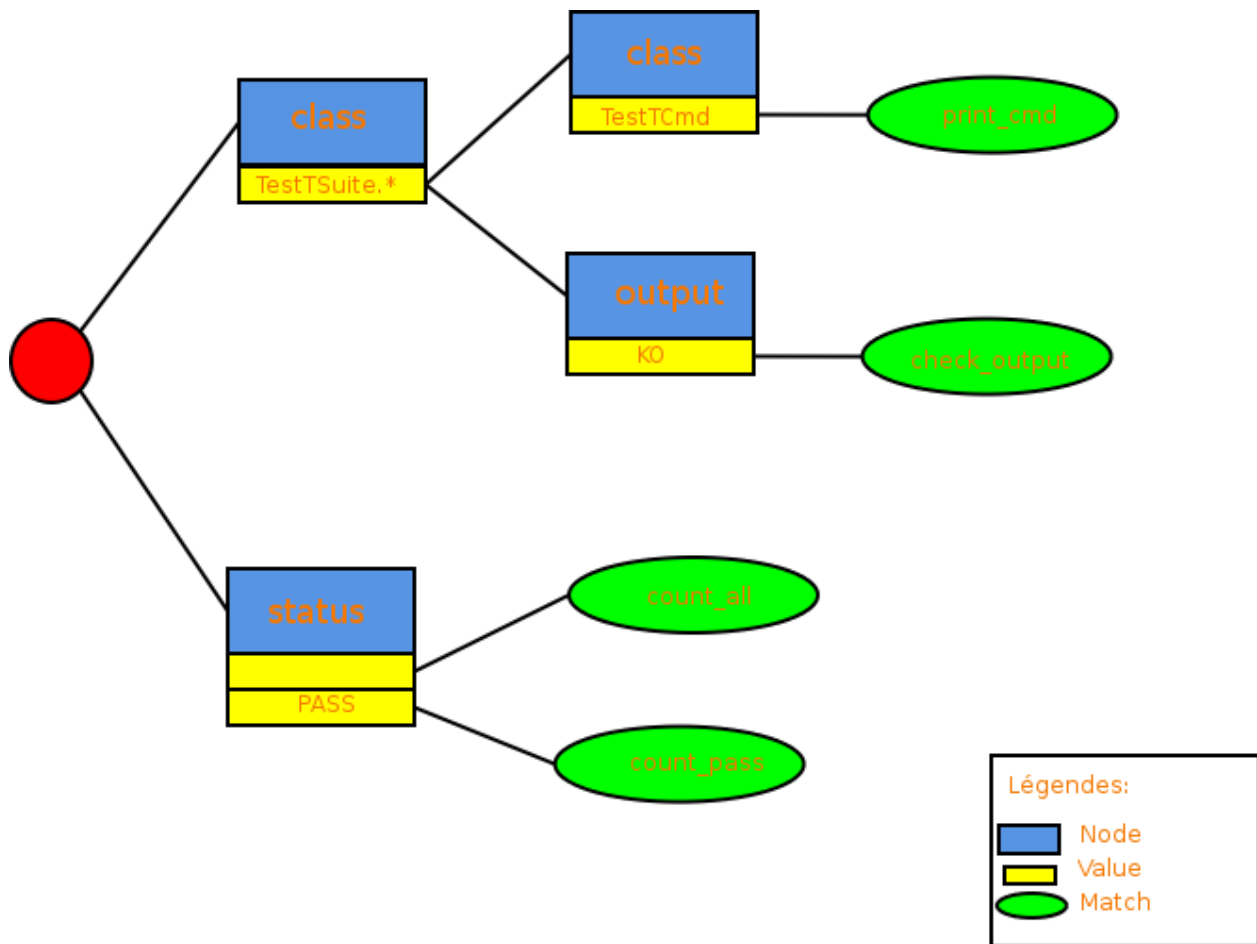


FIG. 5.2 – Graphe résultat de la compilation de règles d'un filtre .

on obtient le graphe suivant :

Le graphe ressemble assez à la partie alpha de la représentation interne de RETE, mais sans tenir compte de la longueur des règles. Mais l'algorithme diffère car on va progresser dans le graphe en flux continu. C'est-à-dire qu'on va avoir une liste des noeuds qui sont actifs, et à chaque fois qu'on lit un nouvel élément de l'arbre YAML d'entrée on met à jour les noeuds actifs du graphe. L'action de mise à jour d'un noeud dépend évidemment de son type. Si c'est un noeud Node on regarde si la nouvelle entrée matche le nom et si oui on teste si un des noeuds Value correspond au contenu de la balise. Si un noeud Match est actif on exécute le bloc de commandes sur la nouvelle entrée.

5.1.7 Autres particularités

Le framework propose aussi d'autres fonctionnalités précieuses afin d'atteindre les objectifs de modularité, simplicité d'emploi, et adaptation à tous les besoins. La plupart des ses fonctionnalités nous sont permises uniquement car nous utilisons un langage dynamique et sont parmi les raisons qui nous ont poussés à choisir Ruby comme langage de développement.

Chargement dynamique

Afin de permettre l'ajout de filtres et de tests rapidement, mais aussi d'avoir une exécution rapide, les différents modules sont chargés dynamiquement à l'exécution, et uniquement en cas de besoin.

On pourra ainsi lister les différents modules grâce aux options `-list-tests` ou `-list-filters` mais surtout, pour ajouter sa bibliothèque personnelle de stratégie de test, il suffit d'appeler le programme avec `-I mon_repertoire_de_test`.

Pour ajouter un test on a donc besoin d'écrire le test, de le mettre dans le répertoire d'inclusion, et ensuite on pourra l'appeler directement depuis le fichier de configuration.

Introspection

Une technique fortement liée au chargement dynamique est l'introspection. C'est une possibilité du langage qui nous permet de lister les attributs d'un objet, ce qui paraît évident quand on a compris comment marche une frame. C'est pour cela qu'on pourra dire que Ruby, un langage de script dynamique basé sur la représentation objet, se rapproche très fortement des frames grâce à l'introspection et au système de démons (`if_added`, `if_needed`).

Cette possibilité du langage nous permet de rendre très facile l'ajout de tests, après un chargement dynamique d'un test on peut connaître tous ces attributs et savoir comment on peut les paramétrer, et donc supporter des tests, qui nous sont inconnus, dans notre langage d'entrée.

On propose aussi une aide automatique, car comme nous l'avons vu chaque attribut à un champ description, il suffit alors de parcourir tous les attributs et d'afficher leur descriptions associées.

On a donc un langage objet mais une approche de notre implémentation fortement inspiré des frames.

Distribution du calcul

Les tests sont souvent nombreux et tester un projet peut prendre beaucoup de temps. Mais les tests sont souvent aussi indépendants entre eux. Il est donc intéressant de pouvoir distribuer l'exécution des tests sur plusieurs machines, le framework le permet. Pour cela il suffit de mettre l'attribut `pool` d'une `TestTSuite` à `true`. La `TestTSuite` se dispatchera sur plusieurs machines selon l'implémentation choisie.

Nous avons vu qu'un test se décompose en trois parties : `prologue`, `milogue`, et `epilogue`. Lorsqu'un test est exécuté à distance, c'est en fait le `milogue` qui est exécuté à distance. Il est important de noter ce point pour bien concevoir ces tests, et comprendre les éventuels problèmes. Cela permet aussi lors de l'implémentation d'avoir un partage maximal des données lors du `prologue` et de l'`épilogue`. La communication des données se fait donc à l'intérieur d'un test entre ces différentes étapes.

Communication inter tests

Nous avons dit que tous les tests sont indépendants entre eux. ceci est Vrai dans la plupart des cas, mais parfois il arrive que le résultat d'un calcul fait par un test soit utile dans un test suivant. C'est pour cela qu'on a besoin de pouvoir passer des informations d'un test à un autre. Pour cela on a introduit une table de symbole. Celle ci agit comme une table d'environnement, c'est à dire qu'à chaque nouvelle `TestTSuite`, la table est recopiée et les valeurs acquises sont communes

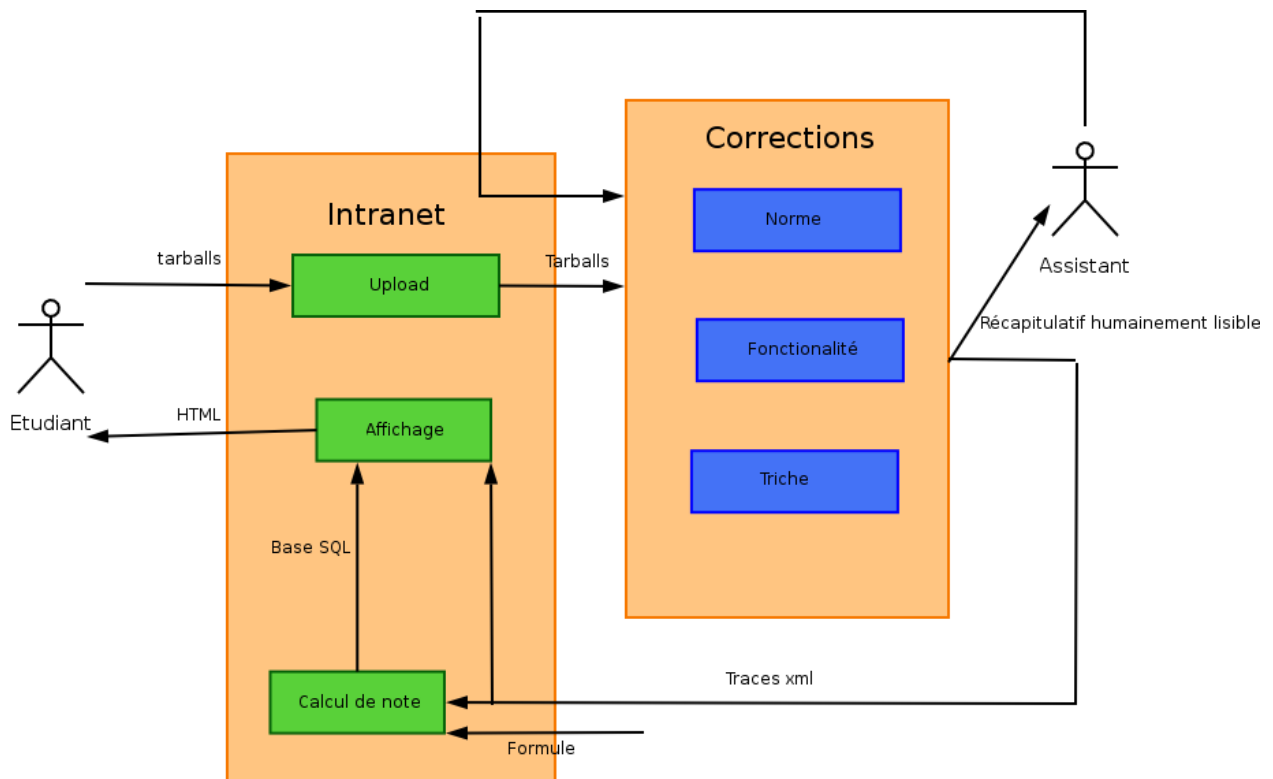


FIG. 5.3 – Détails de la chaîne de notation des assistants.

uniquement au sous arbre en cours. Cette fonctionnalité est très puissante mais périlleuse à utiliser, surtout dans le mode distribué, cela ralentit l'exécution, complexifie le fichier de configuration et peut facilement introduire des bugs d'inattention.

5.1.8 Chaîne d'utilisation

Dans le cadre des assistants, TestTKewl n'est pas une fin en soit mais entre dans un processus de correction des étudiants qui est assez long et complexe. Il met en jeux des éléments extérieurs qui sont :

- les tarballs des étudiants ;
- le calcul d'une note à partir de la trace de moulinette ;
- l'intranet et donc un affichage html ;
- le sujet qui détermine les fonctionnalités du projet de l'étudiant ;
- le rendu par upload des étudiants ;

Le schéma 5.3 montre où s'insère exactement TestTKewl. Malgré toutes ces dépendances externes TestTKewl reste un projet indépendant, totalement utilisable dans un autre contexte, mais assez modulaire pour s'intégrer efficacement.

Certaines contraintes nous étaient imposés dès le départ comme avoir une sortie xml, nécessaire pour la méthode de calcul de note de l'intranet, et puis pour un passage par une xslt pour être affiché proprement en html. Une autre contrainte était de pouvoir tester, non pas un seul programme mais toute une promotion et de garder une trace séparée pour chacun des élèves. Mais toutes les contraintes se résolvent facilement grâce à la modularité à trois niveau de TestTKewl : puissance

d'expression en entrée, possibilités infinie des tests, afficheur complètement paramétrable.

5.1.9 Bilan

En résumé, le framework s'adapte tout à fait à nos besoins, il nous permet l'ajout de tests facilement, ce qui nous permet de tester nos différentes solutions de détection de triche. Grâce aux filtres nous pouvons interpréter les résultats des tests rapidement, facilement et récupérer les informations pertinentes qui feront gagner beaucoup de temps à l'utilisateur.

De plus ce framework étant utilisé dans d'autres circonstances parmi les Assistants, il est maîtrisé par l'équipe, et ainsi de nombreuses autres personnes vont pouvoir contribuer en ajoutant de nouveaux tests. A terme on obtient alors un grand nombre de petits programmes permettant de détecter des cas particuliers de triche, le tout mis ensemble grâce au framework et fournissant à l'utilisateur les informations pertinentes.

Annexe A

Spécifications de la TestKewl

Sommaire

A.1 Description générale du projet	46
A.1.1 Buts	46
A.1.2 Principaux cas d'utilisation de la moulinette	46
A.2 Spécifications générales	47
A.2.1 Fonctionnement général	47
A.2.2 Méthode devant apparaître de base dans la moulinette	48
A.2.3 Cas pratique	49
A.2.4 Modules de sortie	50
A.2.5 Mise en place d'un processus de qualité	50
A.2.6 Coding Style	50
A.2.7 Erreurs	51
A.2.8 Execution de scripts extérieurs	51
A.2.9 Interface avec l'extérieur	52
A.2.10 Options de la ligne de commande	52
A.2.11 Création automatique des références	52
A.2.12 Autres détails et bugs à éviter	53
A.3 Interfaçage avec l'intranet	53
A.3.1 Xml	53
A.3.2 XSL	55
A.4 Glossaire	55

AUTEURS : Marco Tessari et Jerome Pouiller

A.1 Description générale du projet

A.1.1 Buts

1. Générer des fichiers XML interfacable avec l'intranet.
2. Avoir une interface universelle pour toutes les moulinettes (Possibilité d'utiliser des scripts généraux à tous les projets).

3. Permettre l'écriture rapide des moulinettes de tous les projets (Piscine, 42sh, Mini-projets Yacku, Tiger, Exam machine, etc...).
4. Aider l'assistant dans le débogage de la moulinette
5. Fournir l'assistant une bibliothèque de fonction utiles en laquelle il peut avoir toute confiance (Portabilité, tests de cohérence, etc ...)

A.1.2 Principaux cas d'utilisation de la moulinette

- Passage de la moulinette sur toutes les tarball de la promotion (Utilisation dans le script de clustering par exemple)
- Utilisation de la moulinette par le service d'upload
- Passage de la moulinette sur une seule tarball pour la gestion des cas problématique
- Utilisation de la moulinette comme environnement de développement pour la création de batterie de tests pour un projet par un assistant. Ceci inclut, la possibilité de ne passer qu'un test, de tester avec une binaire de référence ou de non-référence ¹(tels que "false" ou "echo Hello")

A.2 Spécifications générales

A.2.1 Fonctionnement général

La principe de fonctionnement de la moulinette est basé sur le chargement dynamique de classes créées par l'Assistant. La moulinette doit fournir à l'assistant des classes par défaut qui, grâce à l'héritage de classes, permettent de servir de bases à la création des classe spécifiques au projet.

Il doit tout d'abord exister une classe Test (représentant un test simple) et une classe TestFactory dérivant toutes les deux d'un TestElement.

TestFactory doit permettre de créer une liste d'Element. Elle permet de créer une liste de Test et de TestFactory.

Lors du démarrage, la moulinette doit rechercher un fichier ² contenant une classe TestFactory initiale qui permettra de faire une liste de TestElement. Pour chaque TestElement de la liste, la moulinette doit l'exécuter si celui-ci est un Test ou obtenir une nouvelle liste si celui-ci est une TestFactory.

La moulinette doit fournir des classes de bases implémentant des méthodes permettant (en gras sur le schéma UML) la création rapide de nouvelles moulinettes.

Par ce mécanisme, l'Assistant a juste besoin de (re)définir la méthode *run()* pour indiquer quel fonctions de tests doivent être utilisés. Par exemple, si le préfère utiliser *diff -u* plutôt que *diff* pour faire ses diff, il lui suffit de surcharger cette méthode.

Ce mécanisme permet de plus de facilement faire évoluer la moulinette. Il suffit de rajouter des méthodes qui pourront ensuite être réutilisée dans d'autres projets.

¹C'est-à-dire qui ne doivent passer aucuns tests

²list.pl, par exemple

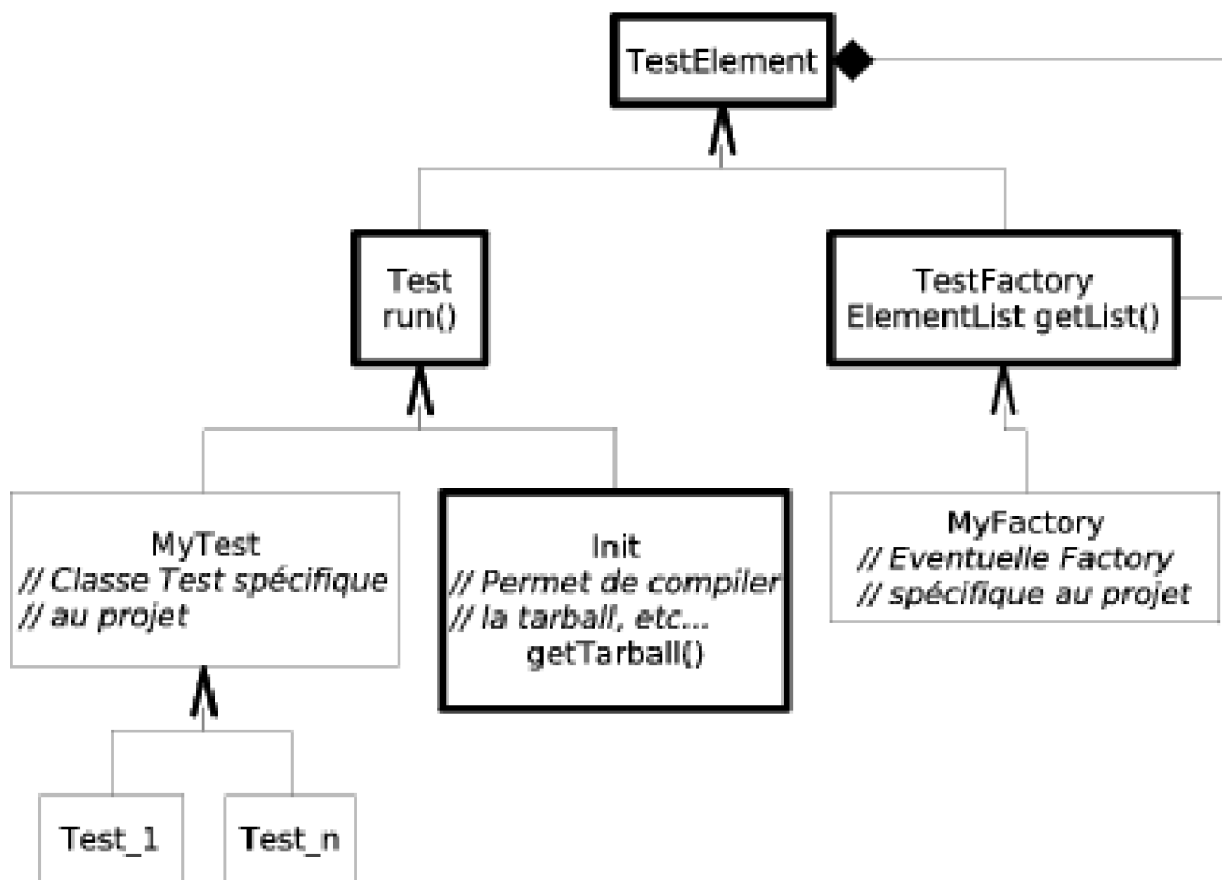


FIG. A.1 – Exemple d’UML de classe de la partie Test de la moulinette. Les classes en gras sont fournies avec la moulinette les autres sont spécifiques au projet et sont créés par l’Assistant.(Cet UML est destiné à expliciter notre besoin. A vous de le modifier à votre convenance.)

A.2.2 Méthode devant apparaître de base dans la moulinette

Exemple de liste de méthodes que l'on aimerait voir apparaître dans la classe TestFactory :

- création de ElementList à partir d'une liste de nom de fichiers implémentant les TestElements.
- création de ElementList à partir d'un fichier contenant la liste des fichiers implémentant les TestElements.

Exemple de liste de méthodes que l'on aimerait voir apparaître dans la classe Test :

- Indiquer la ligne de commande à exécuter.
- Spécifier un fichier devant être redirigé vers l'entrée standard de la binaire de l'étudiant (par défaut, le fichier NomDeLaClasse.in doit être utilisé).
- Spécifier une chaîne de caractères devant être envoyée sur l'entrée standard de la binaire de l'étudiant.
- Spécifier un ou des fichiers avec lequel la sortie standard de l'étudiant doit être comparée (diffée) (par défaut, le fichier NomDeLaClasse.out doit être utilisé)
- Spécifier un ou des fichiers avec lequel la sortie d'erreur de l'étudiant doit être comparée (diffée) (par défaut, le fichier NomDeLaClasse.err doit être utilisé)
- Spécifier un ou des signaux que le la binaire de l'étudiant doit retourner (par défaut SIGTERM)
- Spécifier un ou des codes de sortie que le la binaire de l'étudiant doit retourner (par défaut 0)
- Spécifier une fonction de filtre (*sed*, *tr*, etc...) par lequel doit passer la sortie de l'étudiant avant d'être comparée.
- Spécifier un time out pour l'exécution de la binaire de l'élève
- Retourner la description du test
- Indiquer quelles informations doivent être mise dans la trace xml ou affichées à l'écran (par défaut : la description, le résultat, le nom du test, les résultats des opérations.
- Indiquer quelles informations doivent être affichées aux élèves sur l'intranet (par défaut : la description, le nom du test, les résultats).
- Démarrer un script extérieur (permettant par exemple d'initialiser le test). Une option doit permettre d'indiquer que le code de sortie de ce test doit être pris en compte dans la notation (si la sortie est 0, le test est réussi, sinon, le test est raté).
- Modifier le coefficient du test.

Exemple de liste de méthodes que l'on aimerait voir apparaitre dans la classe Init :

- Utiliser des options passer à la ligne de commande pour (dés)activer certaines options
- copier la tarball dans /tmp
- degunziper
- debzipper
- executer configure
- executer make
- traiter le fichier AUTHORS
- récupérer le login
- récupérer le groupe
- récupérer la trace et la copier dans un repertoire (en prennant en compte les groupes)
- récupérer la trace et la scp sur un compte distant (en prennant en compte les groupes)

- Permettre de facilement tester avec une binaire de référence

N’oublions pas que ces classes ne sont pas destinées à être utilisées telles quelles. Plus le code est modulaire et plus la moulinette fournit de méthodes utilisables par l’Assistant, plus celui-ci sera content.

A.2.3 Cas pratique

Etudions le fonctionnement de la moulinette sur un cas pratique : `root.pl` contient notre Test-Factory initiale. Elle devrait ressembler à³ :

```
package RootFactory extend TestFactory
run () {
  putListFrmName ``test1.pl``;
  push elementList (sub {
    package Test2 extent TestProjet;
    diffOut(``first.out``);
  });
}
```

Lors de l’appel de la méthode `run()` (on aurait pu surcharger la méthode `get()` aussi) de cette classe, la classe contenue dans le fichier `test1.pl` est ajoutée à `ElementList`. On ajoute ensuite la classe `Test2` que l’on déclare en ligne. La moulinette appelle ensuite la méthode `get()` qui lui retourne la liste des `TestElement` et parcourt cette liste.

`test1.pl` pourrait alors contenir :

```
package Test1 extent TestProjet;
run () {
  diffOut();
  diffErrFrmString(````);
}
```

Un fichier `TestProjet` ressemblerait à :

```
package Test\index{projet}Projet extent Test;
diffErrFrmString(String \$str) {
  `echo \$str | diff - \$output`;
}
```

³Le langage bâtarde que j’ai utilisé est, j’espère, assez compréhensible

A.2.4 Modules de sortie

La partie de sortie d'informations (créant le XML par exemple) doit être bien séparée du reste de la moulinette. La moulinette devra posséder un backend modulaire capable, par exemple, de sortir deux traces de type différentes avec un minimum de modification du code. Parmi les modules de sortie que l'on aimerait voir apparaître :

- Sortie Xml
- Sortie sur Stdout
- Sortie sur une fifo pour l'interfaçage avec d'autres programmes (cluster.pl par exemple)
- Sortie vers un module de statistiques

A.2.5 Mise en place d'un processus de qualité

La Moulinette doit être documentée aussi bien du point de vue de l'Assistant que du développeur.

La Moulinette devra posséder une batterie de tests permettant de vérifier la consistance des modifications qui seront faites par la suite ⁴

A.2.6 Coding Style

Vous devez respecter un certain Coding Style :

- 80 colonne (enfin, pas de 200 colonnes...)
- Factoriser le code
- *out* = une sortie standard
- *err* = une sortie d'erreur
- *cod* = une code de sortie
- *sig* = une signal de sortie
- *ext* = une code de sortie d'un script extérieur
- préfixe *s* = fait référence à l'étudiant
- préfixe *r* = fait référence à la référence du projet
- préfixe *pr* = print
- Les interfaces extérieur doivent être en Anglais ⁵
- Tous les répertoire et les fichiers doivent pouvoir être modifié par l'intermédiaire de la ligne de commande ou d'un fichier de configuration⁶

A.2.7 Erreurs

Les lignes affichées par le module de sortie sur Stdout doivent toutes être préfixées par un numéro de trois chiffres (Ceci permet de facilement grepper la sortie)

La norme suivante n'est qu'un exemple vous pouvez la modifier à loisir (c'était la norme utilisée jusqu'à présent).

⁴Le bootstrapping serait une solution élégante

⁵Le code peut être en français, mais, considérez que l'utilisateur est Anglais

⁶Ce pourrait être une bonne idée de pouvoir utiliser le même fichier pour cette configuration que pour définir la classe d'initialisation.

La sortie normal devrait uniquement contenir les messages destinés aux assistants. Une option devrait permettre d'afficher les messages plutôt destinés aux étudiants (3xx). Voici les numéros prévus pour les messages :

DEBUG	1xx	
	11x	Utilisation de la variable <i>xxxx</i>
INFO	2xx	
	21x	Exécution d'un script
	22x	Suppression de fichiers
	23x	Changement de répertoire
STUDENT	3xx	Information pour l'étudiant
	31x	Test
	311	Nom du test
	312	Description du test
	313	Résultat du test
	314	Détails du test
	315	Diff du test
	32x	Résumé
	33x	Entête de la tarball
	331	Environnement
	332	Detarrage
	333	Vérification de Fichiers
	334	Author
	335	configure
	336	make
	34x	Un warning sur la tarball
35x	Un truc vraiment pas bien pour l'élève (ça compile pas, etc...)	
WARNING	4xx	Test de cohérence qui foire
ERROR	5xx	Truc pas normal du tout
	51x	Erreur d'exécution d'un script de la moulinette
	52x	Erreur lors de la suppression d'un fichier
	53x	Erreur lors d'un changement de répertoire
FATAL ERROR	6xx	Erreur empêchant de continuer
INTERNAL ERROR	7xx	Vous venez de trouver un bug

A.2.8 Execution de scripts extérieurs

Il y aura sûrement un certain nombre de scripts (ou de commandes) exécutés dans un sous processus de la moulinette.

Tous les scripts exécutés par la moulinette sont exécutés dans un répertoire temporaire appelé « trash ».

Certaines variables d'environnement doivent être maintenues par la moulinette (dans la mesure du possible évidemment) :

- *\$BIN* : Binaire de l'étudiant
- *\$LOGIN* : Login de l'élève courant
- *\$XMLFILE* : Fichier XML
- *\$TEST* : Test en cours

- `$$SOUT` : Fichier contenant la sortie standard de l'étudiant sur le test en cours
- `$$SERR` : Fichier contenant la sortie d'erreur de l'étudiant sur le test en cours
- `$$SCOD` : Fichier contenant le code de sortie de l'étudiant sur le test en cours
- `$$SSIG` : Fichier contenant le code de sortie de l'étudiant sur le test en cours

A.2.9 Interface avec l'extérieur

Comme nous l'avons déjà précisé, la moulinette doit s'interfacer facilement avec :

- Le service d'upload de l'intranet
- Un script de clustering sur les machine du PIE
- La moulinette de Norme
- La moulinette de Triche
- La moulinette de Ramassage

A.2.10 Options de la ligne de commande

La processus d'initialisation de la moulinette doit être partagé entre la moulinette en elle même et la classe d'initialisation. Pour cette raison, les options de la ligne de commande non utilisées par la moulinette devront être transmises à la classe d'initialisation.

A titre d'exemple, j'ai ici détaillé les options que nous utilisons dans la moulinette actuelle

```
moule [OPTIONS] <rootdir>
-m          Move. Deplacer les \index{tarball}tarball finie dans ./fi
-f          Flush 1. Effacer ./trashdir entre les tarballs.
-o tracedir Output. Specifier un repertoire ou stocker les traces
-i \index{tarball}tarball Input. Specifier une/des tarball. Peut etr
-r reftar   Reference. Damander la creation des references [obsolète]
-t test     Test. Specifier un test a faire passer
-c config   Config. Specifier un fichier de configuration
-e setup    sEtap. Specifier la classe d'initialisation
-v --verbose Verbose. Mode verbeux. Affiche les messages en 3xx
-h --help   Afficher l'écran d'aide
<rootdir>  Designe le repertoire ou se trouve l'arborescence de la
           batterie de test
```

A.2.11 Création automatique des références

L'ancienne version de la moulinette permettait de générer automatiquement les fichiers de références (stdout, code desortie, etc...). Nous pensons que cette fonctionnalité n'a pas sa place dans la moulinette. De plus, la généricité de la moulinette la rend difficile à gérer. Pour ces raisons, nous estimons qu'il vaut mieux créer un script spécifique au projet en cours pour la création de référence. Donc, pas d'implémentation dans la moulinette.

A.2.12 Autres détails et bugs à éviter

- Ne pas oublier de gérer les projets en groupes

- Changer les caractères non imprimables en `\x??` dans CDATA
- Changer les caractères non imprimables en `\x??` avant de faire le diff
- Attention a la gestion des fork et des redirection de sortie
- Permettre l'intégration LD_PRELOAD
- La nouvelle moulinette devrait pouvoir remplacer la tc-check

Nous avons déjà réfléchi sur le langage d'implémentation du projet. Celui-ci doit absolument être orientés objet :

Python Langage adapté à nos besoins mais ne se trouve pas sur toutes les architectures de l'école (il suffit de le compiler pour qu'il soit mis sur les DP)

Ruby Mêmes remarques que pour Python. Il bénéficie en plus d'une grammaire permettant d'écrire des classes (et donc des tests) simplement.

Perl Ressemble à un langage objet. Disponible sur toutes les architectures de l'école (Attention peut-être à la version 5.00 des stations Sun). La grammaire pour la programmation orientée objet est un peu complexe et difficile à appréhender.

Un langage compilé Oblige à compiler la moulinette (et donc les tests). Obligation d'avoir une moulinette par architectures. Permettrais de contourner le problème du chargement dynamique de classe.

Multi-langage avec l'aide d'un protocole tel que Swig Oblige à compiler la moulinette sur toutes les architectures mais une seule fois pour tous les projets. Les tests peuvent être écrit dans n'importe quel langage. Usine à gaz.

A.3 Interfaçage avec l'intranet

La moulinette doit généré un fichier Xml qui ensuite afficher par l'intranet à l'aide d'un parseur XSL. Les format du fichier Xml (et par conséquent du XSL) aura sûrement besoin d'être refait pour la nouvelle moulinette.

A.3.1 Xml

J'ai ajouté ici un exemple des informations que le Xml devrait contenir :

- Environnement
 - Architecture
 - Nom de machine
 - Nom du User
 - Nom de la Tarball
 - Date
- Init
 - Récupération
 - Detarrage
 - Liste des fichiers
 - Configure
 - Make
 - Vérification de l'exécutable

- Testsdire
 - Init
 - Stdout
 - Stderr
 - Code
 - Signal
 - Tests (attributs name et desc)
 - Init
 - Stdout
 - Stderr
 - Code
 - Signal
 - Command
 - Stdout
 - Stderr
 - Code
 - Signal
 - Deinit
 - Stdout
 - Stderr
 - Code
 - Signal
 - SubSubSummary
 - Diff stdout
 - Resultat stdout {Failed/OK}
 - Diff stderr
 - Resultat stderr {Failed/Ok}
 - Resultat code {Failed/Ok}
 - Resultat signal {Failed/Ok}
 - Resultat check {Failed/Ok}
 - Resultat final {Failed/Timeout/Abord/SegFault/Suspicious/Cheater}
 - Deinit
 - Stdout
 - Stderr
 - Code
 - Signal
 - SubSummary
 - Nombre de tests total
 - Nombre de tests passes
 - Nombre de tests ok
 - Note
 - Note max
 - Pourcentage
- Deinit
- Summary
 - Nombre de tests total

- Nombre de tests passés
- Nombre de tests ok
- Note
- Note max
- Pourcentage

A.3.2 XSL

La XSL doit afficher les résultats sous forme arborescente (cf. <http://ts.hercules.com/eng/index.php>). Les Assistants doivent avoir accès à l'ensemble des détails de la trace (diff, sortie, etc...). Les élèves ne doivent pouvoir accéder qu'à certains détails (spécifiés par exemple par la balise *visible = true*).

A.4 Glossaire

Développeur : Développeur de la moulinette

Assistant : Développeur de la batterie de test

Fichier TestElement : Fichier contenant le source d'une classe TestElement

Binaire de l'étudiant : La cible de la moulinette.

Bibliographie

- [1] *plagiarism.org*. <http://plagiarism.org/>. Compagny specialized in plagiarism detection online.
- [2] F. C, *Le code source de half-life 2 leaké!*, Présence PC, (2003). <http://www.presence-pc.com/news/n1720.html>.
- [3] X. CHEN, B. FRANCAIA, M. LI, B. MCKINNON, AND A. SEKER, *Shared information and program plagiarism detection*, tech. rep., University of California, December 2003. <http://monod.uwaterloo.ca/papers/04sid.pdf>. Presents tokens compression algorithms and introduce PPM-9 algorithm.
- [4] P. CLOUGH, *Plagiarism in natural and programming languages : an overview of current tools and technologies*, tech. rep., University of Sheffield, July 2000. http://www.dcs.shef.ac.uk/~cloughie/plagiarism/HTML_Version/index.html. Un document sur le plagiat dans les codes sources et les document écrit (langages naturels). Très superficiel mais, permet d'avoir un point de vue sur le plagiat en général. Le document essaye d'être exhaustif. Le point de vue n'est pas technique.
- [5] P. CUNNINGHAM AND A. N. MIKOYAN, *Using CBR techniques to detect plagiarism in computing assignments.*, tech. rep., 1993. <http://citeseer.ist.psu.edu/cunningham93using.html>.
- [6] J. A. W. FAIDHI AND S. K. ROBINSON, *An empirical approach for detecting program similarity within a university programming environment*, Computers and Education, 11 (1987), pp. 11–19.
- [7] M. J. FOLEY, *Microsoft to release third open-source project*, Microsoft Watch, (2004). <http://www.microsoft-watch.com/article2/0,1995,1652280,00.asp?kc=MWRSS02129TX1K0000535>.
- [8] S. GRIER, *A tool that detects plagiarism in pascal programs*, ACM SIGCSE Bulletin, 13 (1981), pp. 15–20.
- [9] P. HECKEL, *A technique for isolating differences between files*, Communication of ACM, (1978), pp. 264–268.
- [10] R. KARP AND M. RABIN, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development, 31 (1987), pp. 249–260.
- [11] J. KRUSKAL, *An overview of sequence comparison*, in Time Warps, String Edits and Macromolecules : The Theory and Practice of Sequence Comparison, D. Sankoff and J. Kruskal, eds., Addison-Wesley, Reading, Mass., 1983, pp. 1–44.
- [12] R. LEMOS, *Microsoft probes windows code leak*, CNET News.com, (2004). http://news.zdnet.com/2100-3513_22-5158496.html.

- [13] N. SADIRAC, R. POSS, M. BIZON, AND M. TESSARI, *EPITA Coding Style Standard*, EPITA. Also known as La Norme.
- [14] S. SCHLEIMER, D. WILKERSON, AND A. AIKEN, *Winnowing : Local algorithms for document fingerprinting saul schleimer*, tech. rep., MSCS, University Of Illinois. <http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf>.
- [15] S. SHANKLAND, *Sco sues big blue over unix, linux*, CNET News.com, (2003). <http://news.com.com/2100-1016-991464.html>.
- [16] W. TICHY, *The string-to-string correction problem with block moves*, ACM Transactions on Computer Systems, (1984), pp. 309–321.
- [17] K. VERCO AND M. WISE, *Software for detecting suspected plagiarism : Comparing structure and attribute-counting systems*, tech. rep., University of Sydney, 1996. http://www.bio.cam.ac.uk/~mw263/ftp/doc/yap_vs_acm.ps.
- [18] M. WISE, *String similarity via greedy strign tiling and running karp-rabin matching*, tech. rep., University of Sydney, 1993. http://www.bio.cam.ac.uk/~mw263/ftp/doc/RKR_GST.ps.
- [19] —, *YAP3 : Improved detection of similarities in computer program and other texts*, SIGCSEB : SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education), 28 (1996). <http://www.bio.cam.ac.uk/~mw263/ftp/doc/yap3.ps>.

Index

42sh, 6, 7

assistant, 3, 4, 7, 8, 10, 32–34, 46–48, 50, 55
assistants, 34

cbr, 11, 32
ctcompare, 10, 11

datamining, 31
distance, 15–19, 22, 23, 29

empreinte, 15
epita, 1, 9, 13, 15, 30

frames, 38, 43

gst, 17, 20, 23

moss, 12

projet, 4–9, 11, 12, 30, 32, 34–36, 44, 46, 47, 49,
53

réseau de neurones, 31
rabin, 20, 21
rendu, 4, 6, 7, 9, 10, 30, 44

tarball, 16, 30, 44, 46, 48
tiger, 6–8
triche, 4, 5, 7–9, 30, 31, 34
tricheur, 9, 31, 32