UNIVERSITÉ DU QUÉBEC À MONTRÉAL

AUTO-DOCUMENTATION ASSISTÉE DE LOGICIELS : GÉNÉRATION ET MAINTENANCE DE FICHIERS README AVEC L'OUTIL NITREADME

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

ALEXANDRE TERRASA

FÉVRIER 2019

REMERCIEMENTS

Je tiens à remercier tout particulièrement mon directeur de recherche, Jean Privat, pour toute son aide et sa patience. Merci pour la réalisation du langage Nit avec lequel j'ai eu le plaisir de m'amuser et de me battre ces dernières années. Merci aussi pour la relecture des 438 «PRs» soumises durant cette thèse.

Guy, mon codirecteur, pour qui un paragraphe ne suffira jamais. Merci pour son soutien, sa direction et ses nombreuses, nombreuses, très nombreuses relectures. Il a su me redonner foi dans les périodes de doutes, sans lui cette thèse n'aurait jamais abouti. Merci encore pour chaque discussion, repas et promenade à St-Jean-de-Matha.

Je ne peux oublier mes parents qui ont toujours été là, malgré l'océan qui nous sépare, durant toute ma scolarité à l'UQAM. Ainsi que mes amis Nath, Paul, Domi, Peggy, Leo, Gen et Camille pour leur gentillesse et leur présence, pour égayer ma vie, merci. Je remercie également mes amis et collègues développeurs du langage Nit. Ils ont eu l'amabilité de me partager leur savoir et leur talent ; Romain, Philippe, Lucas, Alexis et Julien.

Pour Milène qui m'a rappelé de manger, boire et dormir durant ces derniers mois de rédaction. Merci.

Ce travail n'aurait pas été possible sans la contribution du laboratoire LATECE via son financement des projets patrimoniaux et sans le support financier de mon codirecteur, le professeur Guy Tremblay, via sa subvention à la découverte du CRSNG (#183776-2012).

Enfin, merci au mot *documentation* qui apparaît 792 fois dans cette thèse.

TABLE DES MATIÈRES

LISTE DES FIGURES viii

LISTE DES TABLEAUX xiv

LISTE DES LISTINGS xvii

RÉSUMÉ xviii

INTRODUCTION 1

I Documentation et fichiers README **7**

CHAPITRE I
CARACTÉRISTIQUES D'UN BON FICHIER README : ÉTAT DE L'ART 8

1.1 Caractéristiques d'une bonne documentation 11
1.1.1 Contenu de la documentation 12
1.1.2 Synchronisation de la documentation avec le code 13
1.1.3 Disponibilité de la documentation 14
1.1.4 Structure de la documentation 15
1.1.5 Utilisation d'exemples 16
1.1.6 Utilisation d'abstractions 17
1.2 De la bonne documentation au bon fichier README 18
1.3 Recommandations sur le contenu d'un bon fichier README 19
1.4 Travaux antérieurs sur l'analyse de fichiers README 21

CHAPITRE II
UNE ANALYSE EMPIRIQUE DU CONTENU DE FICHIERS README PROVENANT DE GITHUB 23

2.1 Composition du corpus GitHub 24
2.2 Utilisation du fichier README par rapport aux autres supports 27
2.3 Format des fichiers README 28
2.4 Taille des fichiers README 30
2.5 Structuration des fichiers README 32
2.5.1 Profondeur des en-têtes 33
2.5.2 En-tête de premier niveau 36
2.5.3 Catégories des sections 38
2.5.4 Ordre des sections 42

2.6 Utilisation de blocs Markdown 44
2.6.1 Utilisation de blocs de citation 45
2.6.2 Utilisation de blocs de listes 45
2.6.3 Utilisation de blocs de code 46
2.7 Utilisation de constructions Markdown en ligne (*inline*) 48
2.7.1 Utilisation de liens 49
2.7.2 Utilisation d'images 49
2.7.3 Utilisation de mises en relief de texte 50
2.7.4 Utilisation de mises en relief de code 51
2.8 Menaces à la validité de notre étude 51
2.9 Conclusion 53

CHAPITRE III
ASSISTANCE À L'ÉCRITURE DE FICHIERS README ET DE DOCUMENTATION D'API : ÉTAT DE L'ART 56
3.1 État de l'art des générateurs de fichiers README 57
3.1.1 Dimensions de l'étude 58
3.1.2 Analyse des sept outils sélectionnés 59
3.1.3 Conclusion sur les générateurs de fichiers README 67
3.2 État de l'art des générateurs de documentation d'API 69
3.2.1 Structuration 75
3.2.2 Présentation du code source et d'exemples 78
3.2.3 Utilisation d'abstractions 80
3.2.4 Vérification de la qualité 83
3.2.5 Conclusion sur les générateurs de documentation d'API 84
3.3 Vers un meilleur support pour la génération des fichiers README . . . 86

CHAPITRE IV
UNE EXPÉRIMENTATION SUR LA RÉDACTION AUTOMATIQUE DE FICHIERS README 89
4.1 Sélection de contenu automatisée 90
4.1.1 Expérience réalisée 91
4.1.2 Résultats 93
4.1.3 Discussion sur les limites d'une sélection automatisée 96
4.2 Regroupement de contenu automatisé 97
4.2.1 Expérience réalisée 97
4.2.2 Résultats 98
4.2.3 Discussion sur les limites d'un regroupement automatisé . . . 102
4.3 Ordre du contenu automatisé 102
4.3.1 Expérience réalisée 103
4.3.2 Résultats 110
4.3.3 Discussion sur l'ordre du contenu automatisé 117

4.4 Menaces à la validité 118
4.5 Limites d'une approche automatisée : vers une approche semi-automatisée 119

II Documentation d'API pour Nit 122

CHAPITRE V
DOC_DOWN – UNE SPÉCIFICATION ET DES OUTILS DE DOCUMENTATION D'API POUR NIT 123
5.1 Uniformiser la structure des projets avec `nitpackage` 125
5.1.1 `src` – Répertoire des sources Nit 126
5.1.2 `package.ini` – Fichier de description du package 127
5.1.3 `man` – Documentation des exécutables Nit 129
5.1.4 `examples` – Exemples d'utilisation exécutables 131
5.1.5 `tests` – Tests unitaires 132
5.1.6 `nitpackage` un outil pour uniformiser la structure des projets 133
5.2 Documenter les entités Nit avec `nitmd` 134
5.3 Encourager l'écriture d'exemples avec `nitunit` 136
5.4 Synchroniser l'information avec les directives `nitiwiki` 137
5.5 Faciliter les références aux entités de l'API avec `nitindex` 140
5.6 Encourager l'utilisation d'abstractions avec `nituml` 142
5.7 Conclusion 144

CHAPITRE VI
DOC_TOOLS – UN CADRE DE DÉVELOPPEMENT POUR LES OUTILS DE DOCUMENTATION NIT 146
6.1 `nitdoc` – Générateur de documentation d'API du langage Nit 148
6.2 `nitweb` – Serveur de documentation du langage Nit 152
6.3 `nitx` – Outil de documentation en ligne de commande 153
6.4 `doc_commands` – Directives de documentation pour Nit 155
6.5 Bibliothèques complémentaires pour les outils de documentation . . . 162
6.6 Comparaison avec les autres générateurs de documentation d'API . . 162
6.7 Conclusion 164

III Rédaction assistée de fichiers README pour Nit 166

CHAPITRE VII
NITREADME : UN OUTIL POUR ASSISTER LA RÉDACTION DE FICHIERS README 167
7.1 `nitreadme` – Utilisation depuis la ligne de commande 169
7.1.1 `--scaffold` – Échafaudage d'un nouveau README 171
7.1.2 `--enhance` – Suggestion du contenu 173

7.1.3 `--check` – Vérification du contenu 173
7.1.4 `--render` – Compilation du fichier README 174
7.2 `nitweb` – Utilisation de `nitreadme` par son interface graphique 175
7.2.1 `nitweb/readme` – Assistant de rédaction 175
7.2.2 `nitweb/readit` – Assistant de lecture 177
7.3 Suggestions de cartes de documentation 179
7.3.1 Cartes d'échafaudage 182
7.3.2 Cartes de documentation d'API 189
7.4 Conclusion 194

CHAPITRE VIII
`DOC_ALIGN` : UNE BIBLIOTHÈQUE POUR ALIGNER LE CONTENU DU README AVEC CELUI DE L'API 197
8.1 État de l'art de l'alignement entre le code source et la documentation . 201
8.1.1 Alignement de la langue naturelle 201
8.1.2 Alignement des identifiants 202
8.1.3 Alignement des extraits de code 203
8.1.4 Le cas des fichiers README Nit 204
8.2 Corpus de fichiers README du langage Nit 205
8.2.1 Structure des fichiers README du corpus 207
8.2.2 Utilisation des extraits de code 209
8.2.3 Utilisation de la mise en relief de code 210
8.2.4 Conclusion sur les caractéristiques du corpus 212
8.3 Analyse du code de l'API et création du modèle Nit 213
8.4 Indexation du modèle Nit avec `doc_index` 214
8.4.1 Indexation des noms des entités 217
8.4.2 Indexation des commentaires 219
8.4.3 Indexation du code source, des exemples et des tests 222
8.5 Analyse du fichier README et création du modèle Markdown 226
8.6 Alignement du contenu du fichier README avec les entités Nit et validation 227
8.6.1 Alignement des thèmes 227
8.6.2 Alignement des mises en relief de code 233
8.6.3 Alignement des identifiants trouvés dans le texte 238
8.6.4 Alignement des extraits de code 243
8.6.5 Alignement de la langue naturelle 248
8.6.6 Consolidation de l'alignement 251
8.7 Menaces à la validité de notre étude 253
8.8 Conclusion 254

CHAPITRE IX
`DOC_SUGGEST` : UNE APPROCHE POUR SUGGÉRER LES CARTES DE DOCUMENTATION DES FICHIERS README 256

9.1 État de l'art de la suggestion de documentation 259
9.2 Processus de suggestion interactif 261
9.2.1 Suggérer les cartes d'échafaudage 262
9.2.2 Suggérer les cartes de documentation d'API 264
9.3 Processus *a priori* de génération d'un nouveau fichier README 269
9.4 Processus *a posteriori* d'amélioration d'un README existant 270
9.5 Validation 271
9.5.1 Contenu des fichiers générés 272
9.5.2 Comparaison des fichiers générés 274
9.5.3 Résultats de la soumission aux experts 278
9.5.4 Questionnaire de satisfaction 281
9.5.5 Menaces à la validité de notre étude 281
9.6 Conclusion 284

CONCLUSION 285

ANNEXE A
ANALYSE EMPIRIQUE DU CONTENU DE FICHIERS README PROVENANT DE GITHUB 290
A.1 Utilisation de blocs Markdown 290
A.1.1 Utilisation de blocs de citation 290
A.1.2 Utilisation de blocs de listes 292
A.1.3 Utilisation de blocs de code 297
A.2 Utilisation de la syntaxe en ligne 303
A.2.1 Utilisation de liens 303
A.2.2 Utilisation d'images 305
A.2.3 Utilisation de mises en relief de texte 306
A.2.4 Utilisation de mises en relief de code 310

ANNEXE B
DIRECTIVES DE DOCUMENTATIONS NIT 312

ANNEXE C
CAPTURES D'ÉCRAN DES OUTILS DE DOCUMENTATION D'API NIT 315

ANNEXE D
INTERFACE DE L'OUTIL `GOOGLE SPREADSHEET EXPLORE` 319

ANNEXE E
CARTES DE DOCUMENTATIONS SUGGÉRÉES PAR `NITREADME` . . . 320

ANNEXE F
PACKAGES NIT UTILISÉS POUR LA VALIDATION DE `NITREADME` . 327

BIBLIOGRAPHIE 328

LISTE DES FIGURES

Figure Page

1.1 Capture d'écran du fichier README du projet **markdown** une fois compilé vers HTML par GitHub, tel qu'affiché sur la page d'accueil du dépôt de sources du projet. 10

2.1 Nombre d'occurrences de chaque langage d'implémentation des projets de notre corpus. 27

2.2 Supports de documentation utilisés par les projets de notre corpus. 29

2.3 Taille des projets (nombre moyen d'octets) en fonction des listes de projets de notre corpus. 31

2.4 Nombre moyen de lignes des fichiers README en fonction des listes de projets de notre corpus. 32

2.5 Exemple de Markdown avec en-têtes et paragraphes. 33

2.6 Trois exemples d'arbres correspondant aux caractéristiques moyennes des fichiers README de notre corpus. 34

2.7 Arbre représentant la table des matières de la présente thèse. . . . 35

2.8 Nombre d'occurrences de chaque niveau d'en-tête détecté dans notre corpus. 37

2.9 Nombre d'occurrences de chaque niveau d'en-tête détecté en fonction des listes de projets de notre corpus. 38

2.10 Comparaison des fréquences d'apparition des thèmes dans le corpus d'Ikeda *et al.* (2018) par rapport aux thèmes trouvés dans notre corpus et aux projets JavaScript de notre corpus. 40

2.11 Comparaison des fréquences d'apparition des thèmes dans le corpus de Prana *et al.* (2018) par rapport aux thèmes trouvés dans notre corpus. 41

2.12 Nombre moyen de sections par thème en fonction des listes de projets de notre corpus. 42

2.13 Position moyenne des thèmes isolés par Prana *et al.* (2018). 43

2.14 Position moyenne des thèmes isolés par Ikeda *et al.* (2018). 44

3.1 Capture d'écran de la liste de propriétés générée par Javadoc pour la classe `ArrayList` du listing 3.1 71

4.1 Format des listes d'entités fournies aux experts. 93

4.2 Pourcentage des propriétés supprimées par chaque expert en fonction du type. 94

4.3 Pourcentage des propriétés supprimées par chaque expert en fonction de la visibilité. 95

4.4 Nombre de groupes créés par chaque expert. 98

4.5 Pourcentage d'entités placées dans des groupes pour chaque expert. 99

4.6 Nombre moyen d'entités par groupe pour chaque expert. 100

4.7 Nombre de groupes créés pour chaque niveau de section. 101

4.8 Similarités moyennes obtenues sur les ordres des modules avec les distances de Levenshtein et Kendall-Tau. 111

4.9 Similarités moyennes obtenues par les variations de PageRank sur les ordres des modules avec les distances de Levenshtein et Kendall-Tau. 113

4.10 Similarité moyenne des ordres des classes selon les distances de Levenshtein et Kendall-Tau. 114

4.11 Similarité moyenne des ordres des propriétés selon les distances de Levenshtein et Kendall-Tau. 115

4.12 Exemple de comparaison d'arbre avec *Tree Edit Distance*. 117

4.13 Similarité moyenne des ordres complets selon *Tree Edit Distance*. 118

5.1 Place des modules `markdown` et `doc_down` dans l'écosystème de documentation Nit. 124

6.1 Place des outils `nitdoc`, `nitweb`, `nitx` et des modules `doc_tools` et `doc_commands` dans l'écosystème de documentation Nit. 147

6.2 Capture d'écran de la page de présentation du package `markdown` telle que générée par `nitdoc`. 151

7.1 Place de l'outil `nitreadme` dans l'écosystème de documentation Nit. 168

7.2 Processus d'utilisation de l'outil `nitreadme` par la ligne de commande et par le module d'extension au serveur de documentation `nitweb`. 170

7.3 Extrait du squelette de fichier `README.doc.md` généré par la commande `nitreadme --scaffold` pour le package `markdown`. 172

7.4 Interface utilisateur de l'outil de documentation assistée `nitreadme`. 176

7.5 Exemple de carte d'importation du commentaire de la classe `MdParser`. 178

7.6 Exemple de carte de suggestion présentant les actions `Insérer` (*Insert*) et `Ignorer` (*Dismiss*) ainsi que le contenu Markdown inséré dans le document README. 180

7.7 Exemple de fenêtre modale de configuration pour la carte de lien. . 181

7.8 Diagramme de classes des cartes de documentation suggérées par `nitreadme`. 181

7.9 Exemple de suggestion de titre et de description succincte pour le projet `markdown`. 183

7.10 Exemple de carte suggérant à l'écrivain d'ajouter le fichier de description du package (`package.ini`) dans son projet. 183

7.11 Exemple de carte de démarrage (*Getting Started*) pour le projet `markdown`. 185

7.12 Exemple de contenu inséré dans le document pour la carte de démarrage (*Getting Started*) du projet `markdown`. 186

7.13 Exemple de carte de liste de propriétés issues de la classe `HtmlRenderer`. 191

7.14 Exemple de carte d'exemple pour la classe `HtmlRenderer`. 192

7.15 Exemple de carte de diagramme de classes UML pour la classe `markdown::MdRenderer`. 193

8.1 Place des modules `doc_index` et `doc_align` dans l'écosystème de documentation Nit. 198

8.2 Vue générale du processus d'alignement du contenu des fichiers README avec les entités de l'API Nit. 199

8.3 Comparaison du nombre de lignes des fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2. 207

8.4 Comparaison du nombre d'en-têtes de sections dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2. 208

8.5 Comparaison du niveaux des en-têtes de sections dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2. 208

8.6 Comparaison du nombre de paragraphes dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2. 209

8.7 Comparaison du nombre d'extraits de code dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2. 210

8.8 Comparaison du nombre de mises en relief de code dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2. 211

8.9 Processus d'analyse des sources Nit pour la création du modèle. . 213

8.10 Représentation de la classe `Array` sous la forme d'un vecteur indexable.215

8.11 Représentation des entités Nit sous la forme de vecteurs. 215

8.12 Processus d'indexation des noms canoniques, courts et lemmatisés des entités de l'API Nit. 218

8.13 Processus d'indexation des entités du modèle par leurs commentaires.219

8.14 Processus d'indexation des extraits de code. 224

8.15 Processus d'analyse du fichier README en cours d'écriture pour la génération du modèle Markdown. 226

8.16 Illustration des contextes dans le modèle du document. 228

8.17 Pourcentage moyen de blocs Markdown identifiés grâce aux approches d'alignement avec les entités de l'API. 252

9.1 Place du module `doc_suggest` dans l'écosystème de documentation Nit. 257

9.2 Processus de suggestion des cartes de documentation. 258

9.3 Illustration des blocs considérés pour la génération des suggestions des cartes de documentation. 261

9.4 Heuristique de transformation de la structure du package en suite de sections Markdown pour la documentation de l'API. 265

9.5 Comparaison du nombre de lignes moyen des fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub. 275

9.6 Comparaison du nombre d'en-têtes de sections dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub. 276

9.7 Comparaison du nombre de blocs de code dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub. 277

9.8 Comparaison du nombre de mises en relief de code dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub. 278

9.9 Pourcentage de fichiers README générés par `nitreadme --scaffold` et améliorés par `nitreadme --enhance`, qui ont été acceptés, acceptés après modification mineure ou refusés. 279

9.10 Pourcentage de carte de documentation d'API qui ont été acceptées ou refusées (pour diverses raisons) pour `nitreadme --scaffold`. . 279

9.11 Pourcentage de carte de documentation d'API qui ont été acceptées ou refusées (pour diverses raisons) pour `nitreadme --enhance`. . 280

A.1 Bloc de citation transformé en HTML par la plate-forme GitHub. 292

A.2 Fréquence moyenne d'apparition du bloc de citation dans les fichiers README en fonction de la liste d'origine des projets. 293

A.3 Nombre d'occurrences de chaque taille de liste trouvée dans le corpus de fichiers README. 293

A.4 Nombre d'occurrences des langages indiqués dans les blocs de code avec barrières du corpus de fichiers README. 299

A.5 Fréquence du nombre de lignes par bloc de code dans la totalité du corpus de fichiers README. 300

A.6 Fréquence d'apparition de chaque thème de bloc de code en fonction des listes de projets de notre corpus. 301

A.7 Fréquence d'apparition de chaque thème de bloc de code par rapport au type de projet. 303

C.1 Résultat de la commande `children: markdown::MdRenderer` tel que présenté par `nitx`. 315

C.2 Capture d'écran de la page d'accueil de la documentation générée par `nitdoc` avec son catalogue de packages. 316

C.3 Capture d'écran de l'interface d'utilisation HTML du serveur de documentation `nitweb`. 317

C.4 Capture d'écran de l'interface d'utilisation de l'outil `nitweb/readit`. 318

D.1 Cartes de suggestions présentées par l'outil `Google SpreadSheet Explore`. 319

E.1 Exemple de carte suggérant à l'écrivain d'ajouter la clé `package.desc` dans le fichier de package (`package.ini`) de son projet. 320

E.2 Exemple de carte suggérant à l'écrivain d'ajouter un fichier `man` pour l'exécutable `nitmd`. 320

E.3 Exemple de carte de suggestion de sommaire (*Summary*) pour le projet `markdown`. 321

E.4 Exemple de carte de problèmes et support (*Issues*) du projet `markdown`. 322

E.5 Exemple de carte d'auteurs (*Authors*) du projet `markdown`. 323

E.6 Exemple de carte de contribution (*Contributing*) pour le projet `markdown`. 324

E.7 Exemple de carte de tests (*Testing*) pour le projet `markdown`. . . . 325

E.8 Exemple de carte de licence (*License*) du projet `markdown`. 326

LISTE DES TABLEAUX

Tableau Page

2.1 Fréquence d'apparition des thèmes détectés dans les documents du corpus d'Ikeda *et al.* (2018). 39

3.1 Générateurs de fichiers README étudiés. 58

3.2 Comparaison des outils de génération de fichiers README. 68

3.3 Générateurs de documentation d'API étudiés. 73

3.4 Comparaison de générateurs de documentation d'API selon les fonctionnalités utiles à la génération de fichiers README. 85

4.1 Détails des bibliothèques sélectionnées. 92

5.1 Fichiers et répertoires d'un projet selon la spécification de la documentation Nit. 126

5.2 Détail des clés utilisées dans un fichier `package.ini`. 128

5.3 Résultats de l'exécution de trois outils sur les suites de tests de la spécification CommonMark. 136

6.1 Directives de documentation proposées par le cadre de développement `doc_commands`. 156

6.2 Interfaces de manipulation des directives de documentation. . . . 159

6.3 Formats de rendu des directives de documentation. 160

6.4 Bibliothèques additionnelles développées dans le cadre de cette thèse. 161

6.5 Comparaison des fonctionnalités offertes par `nitdoc` par rapport aux autres générateurs de documentation d'API. 163

7.1 Comparaison de `nitreadme` avec les outils de génération de fichiers README. 195

8.1 Description des packages composant le corpus des fichiers README de packages Nit. 206

8.2 Calcul de la fréquence de chaque terme, de la fréquence de document inverse et de *TF-IDF* pour les trois commentaires de notre exemple. 223

8.3 Descripteurs d'actions utilisés dans les vecteurs de code. 225

8.4 Rappel et précision de l'algorithme de sélection des thèmes utilisés dans les blocs Markdown. 232

8.5 Performances de l'algorithme de sélection des entités de l'API depuis les références des mises en relief de code. 237

8.6 Performances de l'algorithme d'alignement grâce aux identifiants dans le texte. 242

8.7 Performances de l'algorithme de sélection des entités de l'API depuis les extraits de code. 246

8.8 Performances de l'algorithme de sélection des exemples de l'API depuis les extraits de code. 247

8.9 Performances de l'algorithme d'alignement de la langue naturelle. 250

8.10 Pourcentage moyen de blocs Markdown identifiés grâce aux approches d'alignement avec les entités de l'API. 252

9.1 Occurrences de chaque section d'échafaudage insérée par les experts (colonne **experts**) ou générée par l'outil `nitreadme` (colonnes **--scaffold** et **--enhance**). 273

9.2 Occurrences de chaque carte de documentation d'API insérée par les experts (colonne **experts**) ou générée par l'outil `nitreadme` (colonnes **--scaffold** et **--enhance**). 274

9.3 Questions posées aux experts après leur session d'utilisation du mode de suggestion interactif de l'outil `nitreadme`. 282

A.1 Taxonomie des thèmes de blocs de citation les plus fréquents dans les fichiers README. 291

A.2 Taxonomie des thèmes de listes les plus fréquents dans les fichiers README. 295

A.3 Taxonomie des thèmes des blocs de code les plus fréquents dans les fichiers README. 298

A.4 Taxonomie des thèmes de liens les plus fréquents dans les fichiers README. 302

A.5 Taxonomie des thèmes d'images les plus fréquents dans les fichiers README. 305

A.6 Taxonomie des thèmes de mises en relief fortes les plus fréquents dans les fichiers README. 308

A.7 Taxonomie des thèmes de mises en relief légères les plus fréquents dans les fichiers README. 309

A.8 Taxonomie des thèmes de mises en relief de code les plus fréquents dans les fichiers README. 310

B.1 Directives de documentation d'API. 312

B.2 Directives pour la structuration de la documentation. 312

B.3 Directives de listes et d'abstractions. 313

B.4 Directives liées à la documentation des packages. 314

F.1 50 packages Nit dont le fichier README a été généré par `nitreadme` avec l'option `--scaffold`. 327

LISTE DES LISTINGS

1.1 Contenu Markdown du fichier README du projet `markdown`. . . . 9

3.1 Code Java (partiel) pour la classe `ArrayList` utilisé pour la génération de la documentation d'API avec Javadoc. 70

5.1 Exemple de fichier `package.ini` pour le package `markdown`. . . . 128

5.2 Exemple de fichier MAN `nitmd.md` pour l'exécutable `nitmd`. 130

5.3 Exemple d'utilisation de l'annotation `is example` sur une méthode de la bibliothèque `markdown`. 131

5.4 Exemple de test unitaire NitUnit pour la bibliothèque `markdown`. . 132

5.5 Exemple de `DocUnit` pour la classe `HtmlRenderer`. 137

5.6 Contenu Markdown du fichier README du projet `markdown` utilisant les directives de documentation Nit. 139

6.1 Exemple d'utilisation de la directive de documentation `CmdChildren`. 157

RÉSUMÉ

Le fichier README est typiquement le premier artefact de documentation vu par les utilisateurs d'un projet logiciel — sa *page de couverture*. Ce fichier permet de présenter les objectifs d'un logiciel, ses fonctionnalités, son utilisation, etc. Selon une étude empirique que nous avons réalisée sur GitHub, la plate-forme de partage de code la plus populaire au monde, 99% des projets contiennent un tel fichier.

Les fichiers README sont écrits selon une structure et un ordre plus ou moins standardisés, et notre étude montre que certaines sections sont communes à la plupart, par ex., installation, signalement de bogues, licence, etc. Étrangement, une part importante des informations rédigées à la main pour la présentation des API — par ex., listes de classes et de propriétés, documentation de ces entités, exemples d'utilisation — correspond à ce qui est habituellement produit de manière automatisée par les outils d'auto-documentation d'API tels que Javadoc ou Doxygen.

Nous proposons donc une approche permettant d'*assister* l'écrivain de fichiers README, tant durant la rédaction que la maintenance. Lors de la rédaction, notre approche consiste à suggérer *des cartes de documentation*, i.e., des extraits de documentation produits par le générateur de documentation d'API pouvant être importés directement dans le corps du README. Puis, durant la maintenance, notre approche assure que le contenu de ces cartes soit tenu synchronisé avec le code source.

La mise en œuvre de notre approche repose sur l'*alignement* du contenu du fichier README avec le contenu de l'API qu'il documente, c'est-à-dire l'établissement de correspondances entre les éléments clés du README et les entités du code source. Ceci permet de sélectionner les cartes de documentation pertinentes à insérer ainsi que les positions où les insérer dans le document.

La principale contribution de notre travail se matérialise dans la conception et la mise en oeuvre d'une suite d'outils pour la production de fichiers README, entre autres, `nitweb/readme`, un outil de rédaction semi-assisté de README faisant des suggestions à l'écrivain de façon interactive, et `nitreadme`, un outil de suggestion pour les fichiers README en ligne de commande pouvant générer de nouveaux fichiers à partir du code (approche *a priori*) ou améliorer des README existants et les tenir synchronisés avec le code (approche *a posteriori*).

Mots clés : documentation logicielle, auto-documentation, API, fichier README, Markdown, alignement.

INTRODUCTION

La documentation est un élément incontournable d'un projet logiciel, plus particulièrement dans le cas de composants réutilisables. Elle est nécessaire au client du composant qui doit apprendre à s'en servir, ainsi qu'au développeur qui doit en maintenir le code.

Le fichier README est le premier artefact de documentation vu par les utilisateurs d'un projet. Ce fichier est considéré aujourd'hui comme la *page de couverture* d'un projet et, comme nous le verrons (chap. 2), il est présent dans presque tous (99%) les projets sur GitHub, la plate-forme de partage de code la plus populaire au monde.

Un fichier README permet aux développeurs de présenter un projet, son objectif, ses fonctionnalités et son utilisation. Dans une étude empirique de ces fichiers sur GitHub (chap. 2), nous montrerons qu'une part importante de leur contenu correspond en fait à l'explication de l'API et de son utilisation.

Les fichiers README sont généralement écrits à la main selon une structure et un ordre plus ou moins standardisés. En effet, on retrouve certaines sections communes à la plupart des fichiers telles la procédure d'installation, le processus de signalement de bogues ou la licence. Toutefois, la partie présentant l'API et son utilisation est généralement écrite selon une approche propre à l'écrivain, suivant un discours pédagogique spécifique à l'API à documenter.

Écrire un fichier README à la main engendre son lot de difficultés. C'est un travail fastidieux et le résultat doit être maintenu synchronisé avec le code. Plusieurs de ces difficultés sont en fait inhérentes à la documentation de logiciels en général (chap. 1).

Pourtant, une part importante de la documentation rédigée manuellement dans le README correspond en fait à ce qui est traditionnellement produit automatiquement par les outils d'auto-documentation d'API tels que Javadoc, Doxygen et autres (chap. 3). En effet, les fichiers README contiennent des listes de classes et de propriétés, comment les utiliser grâce à des exemples d'utilisation, des diagrammes UML, etc.

Les outils d'auto-documentation d'API, aujourd'hui incontournables dans le monde du développement logiciel, sont particulièrement prisés par les développeurs car ils apportent une solution à la synchronisation de la documentation avec le code. Cependant, ces mêmes informations, qui peuvent être extraites depuis le code source et présentées automatiquement par les outils d'auto-documentation, sont utilisées dans les fichiers README mais sont rédigées à la main par l'écrivain. On peut alors se demander pourquoi le fichier README est écrit manuellement et non généré comme l'est la documentation d'API, puisque les informations sont presque les mêmes ?

La structure et le contenu d'un fichier README s'articulent autour d'un discours pédagogique choisi spécifiquement pour le projet à documenter par l'écrivain. Nous montrerons (chap. 4) qu'il est difficile de générer automatiquement cette structure tout en satisfaisant les exigences des écrivains.

Notre proposition est donc de renverser le processus en mettant l'écrivain au centre de la production de documentation et *en l'assistant* pour introduire des morceaux de la documentation d'API, générés automatiquement, directement dans le fichier README. Ceci permet alors de combiner le meilleur des deux mondes : un discours pédagogique, conçu spécifiquement pour les besoins du projet, et une génération automatique, facile à maintenir.

Nous proposons donc une approche *semi-automatisée* d'écriture des fichiers README permettant à l'écrivain d'utiliser la structure qu'il désire mais en y incluant

automatiquement les éléments issus du résultat de l'auto-documentation d'API. Notre approche s'axe sur *la suggestion de cartes de documentation* : des extraits de documentation produits par le générateur de documentation d'API, mais qui peuvent être importés directement dans le corps d'un fichier README. Ces cartes permettent à l'écrivain de présenter des listes de propriétés, des commentaires issus du code, des abstractions telles des diagrammes UML ou encore des exemples, tous des éléments traditionnellement présents dans la documentation d'API.

Notre projet et thèse se placent dans le contexte du développement du langage de programmation Nit [1]. Tout comme son ancêtre PRM (Ducournau *et al.*, 2009; Privat et Ducournau, 2005), Nit est un langage dédié à l'expérimentation de différentes techniques d'implémentation et de compilation. Nit est un langage à objets à typage statique et héritage multiple, dont les fonctionnalités incluent notamment les types nullables (Gélinas *et al.*, 2009), les types virtuels (Torgersen, 1998), la généricité homogène non effacée (Terrasa et Privat, 2013).

Notre choix du langage Nit comme sujet d'étude s'explique par la proximité avec les concepteurs et experts du langage — professeurs et étudiants de maitrise et de doctorat à l'UQAM, ainsi que moi-même. Cette proximité permet une réelle influence sur la spécification du langage et des outils de documentation figurant dans l'écosystème de développement Nit. De plus, le contrôle sur la chaîne de compilation nous permet d'étendre le langage en fonction de nos besoins.

Nit, avant notre travail de thèse, ne disposait que d'un générateur de documentation d'API aux fonctionnalités limitées. Afin de rendre possible notre solution d'assistance à la rédaction de documentation, notre première étape a donc consisté à développer un écosystème de documentation — voir la figure à la page suivante (p. 4) — puis à égaler — et même dépasser — l'état de l'art des systèmes

1. `http://nitlanguage.org`

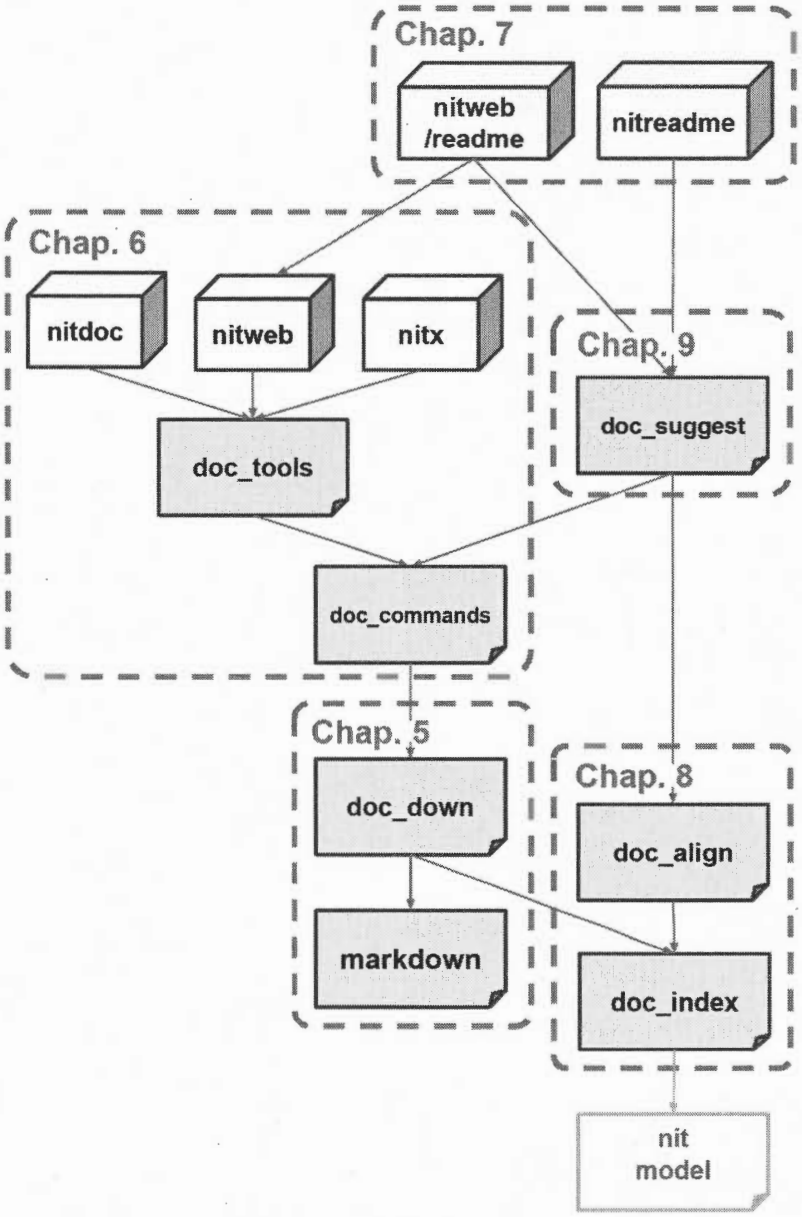


Figure : Aperçu de l'écosystème de documentation développé durant la thèse.

Légende: Les boites blanches sont les outils exécutables. Les cadres grisés correspondent aux modules importés par ces outils. Les encadrés en pointillés représentent la préoccupation principale traitée par chaque chapitre de la thèse.

d'auto-documentation d'API (chaps. 5 et 6) afin de pouvoir ensuite générer le contenu des cartes de documentation.

Une fois ce générateur de documentation d'API développé, nous avons pu ensuite nous atteler au processus de sélection et d'insertion des cartes de documentation dans le contenu des fichiers README.

La principale contribution de notre travail de thèse se matérialise dans la conception (chap. 7) et la mise en œuvre (chaps. 8 et 9) d'une approche — la *suggestion* de cartes de documentation — et d'une suite d'outils pour la production de fichiers README dont :

- `nitreadme` : un outil de suggestion pour les fichiers README en ligne de commande capable de générer de nouveaux fichiers README à partir du code (approche *a priori*) ou d'améliorer des fichiers README existants et de les tenir synchronisés avec le code (approche *a posteriori*).
- `nitweb/readme` : un outil de rédaction de fichiers README semi-assisté faisant des suggestions à l'écrivain, de façon interactive, pendant la rédaction du fichier README, outil présenté sous la forme d'une extension modulaire au serveur de documentation Nit, `nitweb`.

La mise en œuvre de notre approche repose sur l'*alignement* du contenu du fichier README avec le contenu de l'API qu'il documente (chap. 8), c'est-à-dire l'établissement de correspondances entre les éléments clés du README et les entités du code source. Ceci permet alors de sélectionner les cartes de documentation pertinentes à insérer ainsi que les positions où les insérer dans le document (chap. 9).

Les réalisations et contributions de notre thèse sont les suivantes :

- Un état de l'art des caractéristiques d'une bonne documentation et d'un bon fichier README (chap. 1) ;
- Une étude empirique du contenu, de la structure et de l'utilisation de la syntaxe Markdown dans les fichiers README provenant de la plate-forme d'échange de code GitHub (chap. 2) ;
- Un état de l'art des outils de génération de fichiers README et des outils de génération de documentation d'API (chap. 3) ;
- Une étude sur la sélection, le regroupement et l'ordre de présentation des entités de l'API à présenter dans le fichier README (chap. 4) ;
- Une spécification pour la documentation d'API du langage de programmation Nit : `doc_down` et des outils pour supporter cette spécification (chap. 5) ;
- Un cadre de développement d'outils de documentation d'API, un ensemble de bibliothèques supportant la création d'outils de documentation d'API, et une suite d'outils de documentation d'API, tous pour le langage Nit : `doc_tools` (chap. 6) ;
- Une suite d'outils pour la génération semi-assistée et la maintenance de fichiers README : `nitreadme` (chap. 7) ;
- Une approche d'alignement du contenu des fichiers README avec les entités des API qu'ils documentent : `doc_align` (chap. 8) ;
- Une approche de génération de suggestions d'améliorations et de synchronisation avec le code du contenu des fichiers README : `doc_suggest` (chap. 9).

Première partie

Documentation et fichiers README

CHAPITRE I

CARACTÉRISTIQUES D'UN BON FICHIER README : ÉTAT DE L'ART

Un fichier README est un fichier de documentation en format texte associé à un projet. Il permet d'en expliquer les objectifs et l'utilisation aux développeurs souhaitant s'en servir ou le maintenir. L'utilisation de fichiers README est une tradition datant des années 1970 pour les projets ouverts (*open source*) et figure parmi les standards de la fondation GNU (Free Software Foundation, Inc., 2016).

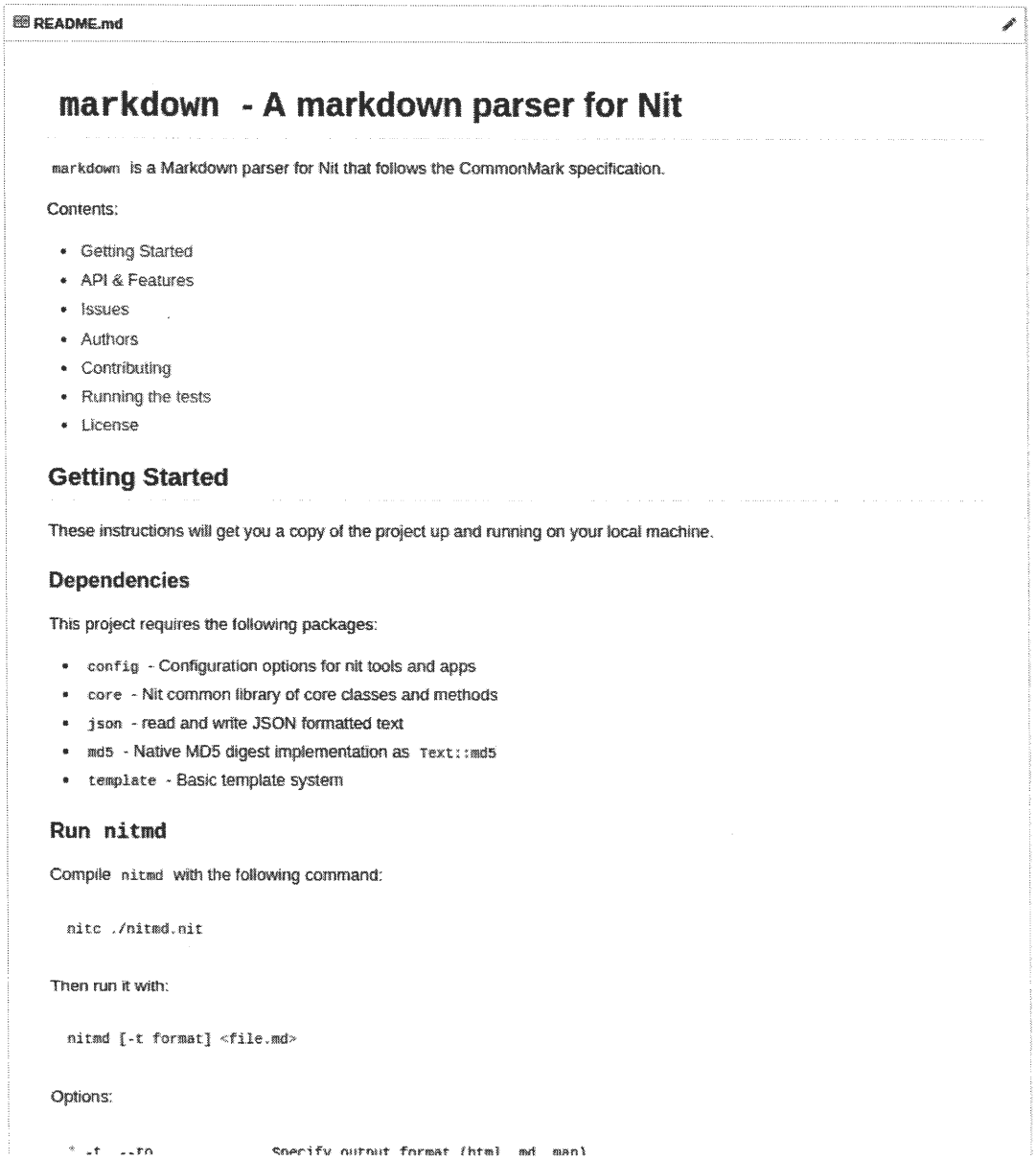
Selon le *General Services Administration* du gouvernement américain (notre traduction) (18F, General Services Administration, 2016a) :

> Chaque dépôt de code doit contenir un fichier README expliquant ce que le projet fait et comment l'installer ou en tester le code. Le README est un élément de documentation important car c'est la première chose que voit un utilisateur en découvrant le dépôt de sources. En écrivant un README lisible, le développeur s'assure que ses collègues et le public seront capables de comprendre les intentions d'un programme, de l'installer et de le modifier pour l'adapter à ses besoins.

Les plate-formes de partage de code populaires telles GitHub, GitLab ou BitBucket encouragent elles aussi l'inclusion d'un fichier README à tous les projets qu'elles

```
# `markdown` - A markdown parser for Nit

`markdown` is a Markdown parser for Nit that follows the
CommonMark specification.

Contents:

 * [Getting Started](#Getting-Started)
 * [API & Features](#API-&-features)
 * [Issues](#Issues)
 * [Authors](#Authors)
 * [Contributing](#Contributing)
 * [Running the tests](#Running-the-tests)
 * [License](#License)

## Getting Started

These instructions will get you a copy of the project up and
running on your local machine.

### Dependencies

This project requires the following packages:

* `config` - Configuration options for nit tools and apps
* `core` - Nit common library of core classes and methods
* `json` - read and write JSON formatted text
* `md5` - Native MD5 digest implementation as `Text::md5`
* `template` - Basic template system

### Run `nitmd`

Compile `nitmd` with the following command:

~~~bash
nitc ./nitmd.nit
~~~

...
```

Listing 1.1: Contenu Markdown du fichier README du projet `markdown`.



README.md

markdown - A markdown parser for Nit

markdown is a Markdown parser for Nit that follows the CommonMark specification.

Contents:

- Getting Started
- API & Features
- Issues
- Authors
- Contributing
- Running the tests
- License

Getting Started

These instructions will get you a copy of the project up and running on your local machine.

Dependencies

This project requires the following packages:

- config - Configuration options for nit tools and apps
- core - Nit common library of core classes and methods
- json - read and write JSON formatted text
- md5 - Native MD5 digest implementation as Text::md5
- template - Basic template system

Run nitmd

Compile nitmd with the following command:

```
nitc ./nitmd.nit
```

Then run it with:

```
nitmd [-t format] <file.md>
```

Options:

```
* -t --to              Specify output format (html, md, man)
```

Figure 1.1: Capture d'écran du fichier README du projet `markdown` une fois compilé vers HTML par GitHub, tel qu'affiché sur la page d'accueil du dépôt de sources du projet.

hébergent et ajoutent que le README permet aussi d'expliquer le processus de contribution (Github, Inc., 2018a). Sur ces trois plate-formes, le README est affiché comme page d'accueil du projet. En guise d'exemple, le listing 1.1 contient un extrait du code source pour le fichier README de la bibliothèque `nit/markdown`, alors que la figure 1.1 montre ce même fichier tel qu'affiché par GitHub.

Avant de proposer une solution visant à assister la rédaction des fichiers README, nous avons cherché à comprendre ce qui compose un tel fichier et quelles sont les bonnes pratiques à suivre. Pour ce faire, nous avons examiné la littérature sur le sujet. Dans la section 1.1, nous résumons tout d'abord les qualités d'une bonne documentation grâce à une revue de littérature sur le sujet. À partir de ces observations et selon les recommandations de différentes autorités dans le domaine, nous tâchons de déterminer les qualités d'un bon fichier README et les bonnes pratiques à suivre pour sa rédaction dans la section 1.3. Enfin, à la section 1.4, nous présentons quelques travaux traitant d'analyses de contenu de fichiers README trouvés sur la plate-forme GitHub.

Dans le chapitre suivant (chap. 2), nous présenterons l'étude que nous avons nous même réalisée sur un corpus de dépôts de code contenant des fichiers README.

1.1 Caractéristiques d'une bonne documentation

De par sa définition, un fichier README permet de documenter comment utiliser un produit logiciel, *donc son API.* Ainsi, comprendre les qualités d'un bon README implique d'abord de mieux comprendre les qualités d'une bonne documentation logicielle. Pour cela, nous avons effectué une revue des études liées à la qualité des supports de documentation pour les API.

Forward et Lethbridge (2002) déterminent, par un sondage réalisé auprès de 48 développeurs, quels sont les attributs d'une documentation qui impactent son efficacité. Selon eux, les six attributs les plus importants aux yeux des développeurs sont les suivants :

1. Le contenu de la documentation
2. Sa synchronisation avec le code
3. Sa disponibilité
4. L'utilisation d'exemples
5. L'organisation du document
6. L'utilisation de diagrammes

Ces résultats sont corroborés par un autre sondage réalisé par Robillard et DeLine (2010), cette fois auprès de 440 professionnels, concernant les limites de la documentation par rapport à l'apprentissage d'une API.

Dans les sous-sections qui suivent, nous détaillons chacune de ces dimensions ainsi que leurs impacts sur la qualité de la documentation, et nous présentons les principales recommandations selon l'état de l'art de compréhension des critères d'une bonne documentation.

1.1.1 Contenu de la documentation

Il va de soi que la pertinence du contenu de la documentation par rapport au projet qu'elle documente est un attribut clé de sa qualité.

Pour Lethbridge *et al.* (2003), la « documentation est généralement mal écrite » et son contenu mal choisi. Les projets contiennent trop de documentation et trouver le contenu utile peut être suffisamment difficile pour décourager les lecteurs.

Robillard et DeLine (2010) relèvent dans leur étude que l'*intention* de la documentation, c'est-à-dire ce que la documentation souhaite effectivement expliquer, est d'une importance capitale pour le lecteur.

La documentation inutile est une réalité dans les projets logiciels si l'on en croit Forward et Lethbridge (2002). Elle représente une perte de temps à la fois pour le lecteur et pour l'écrivain.

Recommandations Pour Forward et Lethbridge (2002), tout nouvel outil de documentation devrait avant tout se concentrer sur l'amélioration du contenu de la documentation.

Selon Berglund (2000), le contenu de la documentation doit s'appuyer sur des scénarios d'utilisation. Ce constat est confirmé par Robillard et DeLine (2010) qui remarquent aussi que le contenu de la documentation doit s'appuyer sur des cas d'utilisation concrets. Wang et Godfrey (2013) ajoutent qu'une documentation contenant des exemples d'utilisation typiques d'un projet augmente la productivité des développeurs.

Nous ajouterions aussi que la documentation devrait posséder un sommaire ou un résumé expliquant son contenu afin de simplifier le travail du lecteur.

1.1.2 Synchronisation de la documentation avec le code

Pour Lethbridge *et al.* (2003), « la documentation est trop souvent périmée ». Une documentation périmée ou qui n'est pas synchronisée avec le code est pire que l'absence de documentation en ce qui concerne la facilité de prise en main d'une nouvelle API (Lutters et Seaman, 2007; Robillard, 2009).

Selon Robillard et DeLine (2010), un contenu de documentation périmé est le second obstacle à la compréhension d'une API après un mauvais contenu. Les indices de désynchronisation d'une documentation avec le code source provoquent une baisse de confiance de la part du lecteur et l'encouragent à se tourner vers une autre source d'information. De plus, Wingkvist et Ericsson (2010) utilisent la désynchronisation de la documentation par rapport au code comme un indicateur d'une documentation de mauvaise qualité.

Recommandations Pour Forward et Lethbridge (2002), une majorité des informations composant une bonne documentation peut être extraite du code source. L'utilisation d'outils d'auto-documentation facilite la maintenance et la tenue à jour de la documentation — contrairement à l'utilisation d'éditeurs de texte ou de logiciels de traitement de texte style «Office». Le résultat produit par les outils automatisés est plus adéquat pour communiquer les vraies fonctionnalités d'un système.

Cependant, les outils d'auto-documentation d'API n'offrent pas une flexibilité suffisante pour documenter tous les aspects d'une API (Roehm *et al.*, 2012). Les écrivains de documentation ont souvent recours à un autre support comme les wikis, les sites internet ou les fichiers README pour documenter les cas d'utilisation ou les décisions de conception.

Un autre désavantage de l'auto-documentation d'API est présenté par Robillard et DeLine (2010), qui constatent qu'une documentation fragmentée rend l'accès à l'information difficile et expliquent qu'une documentation suivant une narration continue est plus efficace.

1.1.3 Disponibilité de la documentation

Il peut être difficile de différencier une documentation introuvable d'une documentation inexistante et le développeur peut perdre un temps précieux à s'en rendre compte. D'après Dagenais et Robillard (2012), toute documentation qui n'est pas directement fournie avec les sources d'un projet est difficile à localiser par le lecteur.

Face à l'absence de documentation officielle, les lecteurs se tournent vers d'autres sources d'information comme les forums ou les blogs (Parnin et Treude, 2011) ce qui favorise la fragmentation de l'information (Hens *et al.*, 2012).

Sans effort de documentation original, le développeur ou écrivain de la documentation perd de l'information sur son propre projet, et donc le coût pour retrouver cette information augmente avec le temps (Vestdam et Nørmark, 2004).

Recommandations Lethbridge *et al.* (2003) voient une explication au manque de documentation par le fait que le processus de rédaction est long et que l'effort engendré n'égale pas les bénéfices apportés.

Klare (2000) définit une bonne documentation comme étant avant tout une documentation simple à écrire et à maintenir. Il mesure l'efficacité de la documentation par rapport au temps alloué à la produire.

Friendly (1995) note que la documentation devrait se trouver au même endroit que les sources du projet pour être utile. Pour Forward et Lethbridge (2002), les outils de documentation devraient faciliter la mise à disposition de la documentation et offrir différentes options de publication. Enfin, Buse et Weimer (2008) rappellent que la présence d'une documentation générée, même limitée, vaut mieux que l'absence de documentation.

1.1.4 Structure de la documentation

Forward et Lethbridge (2002) montrent que la structure de la documentation impacte son efficacité. Une bonne structure est considérée comme un élément clé d'une bonne documentation.

La structure d'un texte, représentée par ses sections et sous-sections, favorise la compréhension du lecteur (Guthrie *et al.*, 2015). Outre le niveau de compréhension, un manque de structure limite aussi la vitesse de lecture (Klare, 2000).

Trop de structure peut aussi nuire au passage de l'information. Parmi les problèmes de structure, Robillard et DeLine (2010) soulignent notamment le fait qu'une documentation fragmentée rend l'accès à l'information difficile.

Recommandations Selon Robillard et DeLine (2010), la taille de la documentation n'est pas un obstacle à l'apprentissage car les développeurs sont habitués à survoler de longs documents à la recherche de l'information désirée. Ils suggèrent donc l'usage d'une narration continue afin de faciliter la navigation et d'offrir un point d'entrée au lecteur. Une structure de documentation proche de celle du code favorise la compréhension.

Cette recommandation n'est pas nouvelle : déjà en 1992 avec le *literate programming*, Knuth (1992) proposait d'écrire la documentation suivant une narration continue organisée à la fois autour d'un discours pédagogique et de la structure du code.

Vestdam et Nørmark (2004) remarquent que certains aspects de la documentation sont transversaux aux préoccupations du code et que la structure du récit doit pouvoir s'adapter à ces besoins. Il faut donc trouver un juste milieu entre une structure proche de celle du code et une structure agréable à lire.

1.1.5 Utilisation d'exemples

Les utilisateurs qui découvrent une bibliothèque utilisent les exemples pour comprendre son fonctionnement (de Souza *et al.*, 2005; Buse et Weimer, 2012; Duala-Ekoko et Robillard, 2012).

Forward et Lethbridge (2002) comptent l'utilisation d'exemples parmi les critères d'une bonne documentation. L'absence d'exemple dans la documentation encourage les lecteurs à se tourner vers d'autres sources d'information (Oney et Brandt, 2012).

Recommandations Hoffman et Strooper (2001) considèrent qu'une bonne documentation doit contenir des exemples de code exécutables. Ils proposent d'enrichir la prose de documentation par des exemples vérifiables afin d'assurer leur synchronisation avec le code.

D'ailleurs, pour Forward et Lethbridge (2002), les outils de documentation devraient faciliter l'intégration des exemples dans la documentation et leur synchronisation. Ils ajoutent qu'il pourrait être utile d'analyser le code des tests unitaires pour améliorer le résultat de l'auto-documentation.

Robillard et DeLine (2010) précisent que les exemples sont plus efficaces lorsqu'ils sont liés à des cas d'utilisation concrets. Ils aident alors le lecteur à comprendre les objectifs et intérêts d'une API. De plus, les exemples concis ont une meilleure efficacité pour expliquer les détails d'une API (Nasehi *et al.*, 2012).
Ainsi, nous résumons les qualités d'un bon exemple de documentation :

- Fonctionnel : s'exécute du premier coup ;
- À jour : correspond à ce qui est documenté ;
- Explicatif : montre une utilisation, un comportement ou une bonne pratique ;
- Minimal : ne montre que le code nécessaire.

1.1.6 Utilisation d'abstractions

D'après Robillard et DeLine (2010), un haut niveau d'abstraction est nécessaire pour expliquer les buts et usages d'une API. Les cas d'usage à documenter sont parfois transversaux aux préoccupations du code.

Pour les lecteurs, la documentation doit permettre de comprendre le fonctionnement d'une bibliothèque à différent niveaux d'abstraction. En fonction du niveau d'expérience du lecteur, différents niveaux de détails sont nécessaires (Deursen et Kuipers, 1999; Buse et Weimer, 2012).

Les lecteurs nécessitent une carte mentale du code pour mieux comprendre les API (Ducasse et Lanza, 2005). En grossissant, les API deviennent de plus en plus difficiles à cerner, la carte mentale devenant de plus en plus difficile à retenir.

Recommandations Les diagrammes et autres éléments visuels de documentation aident le lecteur à saisir plus rapidement le contenu d'une API complexe (Holmes *et al.*, 2006; Zhang *et al.*, 2011; Hao *et al.*, 2013). Ils permettent de présenter visuellement des données de manière simplifiée et structurée, mais doivent être supportés par du texte pour être efficaces (Antoniol *et al.*, 2002; Heijstek *et al.*, 2011).

Selon de Souza *et al.* (2005), les modèles logiques de données et les diagrammes de classes figurent parmi les cinq supports de documentation les plus utiles. Ces deux types de représentations permettent de mieux comprendre les intentions du code.

Pour Deursen et Kuipers (1999), les abstractions devraient être générées à partir du code source afin d'être toujours à jour.

1.2 De la bonne documentation au bon fichier README

Nous pouvons résumer les qualités d'une bonne documentation sur la base du respect des critères suivants :

- Contenu pertinent : en lien avec l'API documentée, basé sur des cas d'utilisation concrets ;
- Disponible : facilement accessible, livrée avec le code source ;
- Structurée : narration continue structurée en fonction du code et des cas d'utilisation ;
- Contenant des exemples : basés sur des scénarios d'utilisation concrets, les exemples sont fonctionnels, à jour, explicatifs et minimaux ;
- Contenant des abstractions : des diagrammes et autres représentations visuelles permettent d'abstraire la complexité d'une API.

Parmi les trois supports externes à la documentation cités par Roehm *et al.* (2012), seul le fichier README apporte une solution à chacun des obstacles signalés dans la revue de littérature que nous avons effectuée. En effet, le README :

— est livré avec les sources du projet ;
— suit une narration continue et sans fragmentation ;
— peut être structuré à l'aide de sections et sous-sections ;
— permet d'expliquer les objectifs d'un projet ;
— permet de présenter des exemples de code et des scénarios d'utilisation.

À l'inverse, avec les wikis et sites internet, la présence d'hyper-liens et l'absence d'un sens de lecture peuvent être déconcertantes pour un lecteur. Le fichier README, malgré son style en format texte et son manque d'abstractions, semble le support idéal pour rédiger le manuel d'utilisation d'un logiciel ou d'une bibliothàque.

1.3 Recommandations sur le contenu d'un bon fichier README

Les qualités d'un bon fichier README sont plus faciles à identifier que celles de la documentation logicielle en général.

Le bureau 18F définit le contenu d'un bon fichier README comme suit (18F, General Services Administration, 2016a) :

— Quel est le dépôt ou le projet ?
— Comment fonctionne-t-il ?
— Comment l'utiliser ?
— Quel est son objectif ?

GitHub propose une version plus détaillée du contenu d'un bon fichier README (Github, Inc., 2018b), inspirée en partie du contenu suggéré par le bureau 18F :

— Nom du projet
— Description
— Table des matières (optionnelle)
— Installation
— Utilisation

— Contribution

— Crédits

— Licence

De nombreux exemples de fichiers README se retrouvent sur GitHub. Ces exemples sont fournis par des utilisateurs et peuvent être réutilisés par d'autres utilisateurs. Bien qu'ils suivent tous la même hiérarchie que celle proposée par GitHub, certaines variations peuvent être observées.

Ainsi, l'exemple fourni par Thompson [1] propose en plus les sections *Déploiement* (*Deployment*), *Dépendances* (*Build with*), *Gestion des versions* (*Versioning*), alors que les sections *Installation* et *Utilisation* sont regroupées sous le parent *Pour débuter* (*Getting started*).

L'exemple de Litt [2] comprend quant à lui les sections *Motivations* (*Background*) et *Badge* (*Badge*) — pour lister les badges GitHub utilisés par le projet [3].

Devant tant de variations, on peut se demander à quoi ressemble la structure d'un fichier README *typique* — «*moyen*» — et quelles sont les sections que l'on peut y trouver. Dans la prochaine section, nous examinons quelques travaux faits sur ce sujet. Puis, dans le prochain chapitre, nous présentons notre propre analyse.

---

1. `https://gist.github.com/PurpleBooth/109311bb0361f32d87a2`

2. `https://github.com/RichardLitt/standard-readme`

3. Les badges (`https://github.com/badges/shields`) sont de petites images ajoutées au début des documents README pour indiquer le statut, la qualité ou tout autre indication sur un projet.

1.4 Travaux antérieurs sur l'analyse de fichiers README

Pour trouver des projets contenant un fichier README, on peut se tourner vers la plate-forme d'hébergement de projets libres GitHub [4]. En 2017, GitHub totalisait 24 millions d'utilisateurs et 67 millions de dépôts Git. La plate-forme a enregistré plus d'un 1 milliard de *commits* au cours de l'année 2016 [5]. Ces statistiques font de GitHub une source d'information incontournable pour tout scientifique souhaitant analyser les habitudes des développeurs (Dabbish *et al.*, 2012; Thung *et al.*, 2013).

L'utilisation des fichiers README trouvés sur GitHub n'est pas une idée nouvelle. Dans le passé, ces fichiers ont notamment été utilisés pour extraire les commandes de compilation de projets Java (Hassan et Wang, 2017), pour extraire le CV de développeurs (Hauff et Gousios, 2015; Greene et Fischer, 2016), pour trouver des similitudes entre projets (Zhang *et al.*, 2017), pour cataloguer les types de projets (Sharma *et al.*, 2017), ou encore pour détecter des problèmes de dépendances de packages (Decan *et al.*, 2016).

D'autres chercheurs ont analysé le contenu des fichiers README trouvés sur GitHub pour en comprendre leur contenu. Ainsi, nous avons trouvé deux études récentes ayant cet objectif, soit celles de Ikeda *et al.* (2018) et de Prana *et al.* (2018).

Ikeda *et al.* (2018) présentent une étude empirique sur le contenu des fichiers README de 43.900 projets JavaScript. Leur objectif est de catégoriser les sections présentes dans les fichiers README afin de mieux comprendre la nature du contenu du fichier. Dans leur étude, les projets sont sélectionnés depuis le gestionnaire de packages JavaScript `npm` [6] puis les fichiers README sont extraits depuis GitHub

4. `http://github.com`

5. `https://octoverse.github.com/`

6. `https://www.npmjs.com/`

quand un lien vers le dépôt est fourni. Les titres de sections sont extraits depuis les fichiers README puis lemmatisés afin de ne garder que la racine des mots composant le titre. Les titres de niveaux supérieurs à deux sont systématiquement ignorés. Les titres sont ensuite catégorisés selon la correspondance des racines par rapport à des thèmes pré-sélectionnés tels *install* ou *author*. Ils créent ainsi un taxonomie de 20 thèmes utilisés dans plus de 1% de leur corpus, lesquels thèmes sont *usage* (26.7% des projets), *install* (26.1%) et *license* (15.9%). L'utilisation de certains thèmes peut dépendre du type de projet. Ainsi, les termes *install* et *license* sont utilisés plus souvent pour des bibliothèques, alors que le terme *option* est plus utilisé pour documenter des applications.

Prana *et al.* (2018) catégorisent le contenu de 393 fichiers README en huit thèmes. Ils sélectionnent les projets au hasard sur GitHub grâce à l'API JSON/REST[7]. Dans leur étude, le contenu des sections est catégorisé manuellement par deux annotateurs selon les thèmes *quoi* (*what*), *pourquoi* (*why*), *comment* (*how*), *quand* (*when*), *qui* (*who*), *références* et *autres*. Leur catégorisation montre que le thème *quoi* est le plus fréquent (97% des documents, 16.8% des sections) suivi par *comment* (88.5% des documents, 58.4% des sections), *références* (60.8% des documents, 20.4% des sections) et enfin *qui* (52.9% des documents, 7.7% des sections). Les documents README semblent donc contenir des explications sur ce que fait le code, comment l'utiliser et qui contacter en cas de besoin.

Dans le prochain chapitre, nous présentons l'étude que nous avons nous-mêmes effectuée sur des fichiers README trouvés sur GitHub, pour mieux en comprendre leur contenu, la structure, le format, etc.

---

7. `https://developer.github.com/v3/`

CHAPITRE II

UNE ANALYSE EMPIRIQUE DU CONTENU DE FICHIERS README PROVENANT DE GITHUB

Dans ce chapitre, nous présentons l'analyse que nous avons effectuée pour identifier les caractéristiques des fichiers README et vérifier s'ils se conforment, ou non, aux bonnes pratiques suggérées pour leur écriture. Notre étude de fichiers README trouvés sur GitHub visait plus particulièrement à répondre aux questions suivantes :

- Est-ce que les projets utilisent effectivement des fichiers README ?
- Quel est le format utilisé pour rédiger ces fichiers ?
- Est-ce que ces fichiers remplissent les critères d'une bonne documentation ?
- Comment le format utilisé aide à remplir ces critères ?
- Quelles sont les pratiques couramment utilisées pour composer un README ?

Notre étude diverge des deux études mentionnées au chapitre précédent (Ikeda *et al.*, 2018; Prana *et al.*, 2018) sur plusieurs points :

- Les projets sont sélectionnés par tranches de popularité sur la plate-forme GitHub ; nous cherchons ainsi à établir un lien entre la composition de la structure et la *notoriété* du projet.
- Nous considérons aussi les en-têtes des sections de niveau supérieur à deux.
- Notre analyse ne s'arrête pas au niveau des sections mais comprend aussi l'étude de l'utilisation de toutes les constructions du langage de présentation le plus utilisé dans les fichiers README, soit le Markdown.

— L'analyse des sections que nous réalisons comprend l'aspect *ordre des sections*, aspect ignoré par les études précédentes. Nous apportons aussi des précisions sur les pratiques entourant le choix des titres associés aux en-têtes de niveau 1 dans les projets.

Dans la section 2.2, nous montrons que les fichiers README sont effectivement utilisés pour documenter des projets réels — projets décrits en section 2.1. De plus, nous précisons en section 2.3 que le format de présentation le plus largement utilisé pour l'écriture de fichiers README sur GitHub est le Markdown, alors que la section 2.4 traite de la taille des fichiers README. Nous analysons ensuite la structure des fichiers README pour déterminer les sections qui les composent et l'ordre de ces sections. Nous déterminons un gabarit de fichier README à partir des données collectées en section 2.5. Puis nous nous penchons sur l'utilisation du format Markdown. La section 2.6 traite de l'utilisation des blocs de présentation Markdown, alors que la section 2.7 traite de l'utilisation des constructions en ligne. Nous utilisons ces résultats pour déterminer les pratiques courantes de l'utilisation du format Markdown dans les fichiers README. Enfin, la section 2.8 traite des menaces à la validité de notre étude et la section 2.9 en présente les conclusions et détermine les objectifs d'un éventuel outil d'aide à la rédaction de fichiers README. Les observations faites dans ce chapitre nous permettront donc de proposer une meilleure solution basée sur des données issues du monde réel.

2.1 Composition du corpus GitHub

La plus grande part des projets a été sélectionnée grâce au moteur de recherche de GitHub. Celui-ci permet de lister les projets par ordre décroissant de nombre d'étoiles (*stargazers count*) : le nombre d'étoiles associées à un projet indique combien d'utilisateurs GitHub « aiment » ce projet.

Bien que le nombre d'étoiles d'un projet ne reflète pas nécessairement la qualité de sa documentation, il permet de juger de sa *notoriété*. Nous nous sommes donc basé sur l'idée qu'un projet connu ou utilisé par un grand nombre de développeurs a plus de chance de contenir une bonne documentation qu'un projet personnel utilisé uniquement par son concepteur. Ainsi, nous avons sélectionné 500 projets selon plusieurs tranches — mutellement exclusives, la borne supérieure étant exclue — de nombre d'étoiles : 10–100, 100–1000, 1000–5000, 5000–10000, 10000 et plus.

Les projets sont sélectionnés grâce à la requête HTTP suivante où `MIN` et `MAX` sont à remplacer respectivement par les bornes inférieure et supérieure du nombre d'étoiles :

```
GET https://api.github.com/search/repositories?q=stars:MIN..
    MAX&sort=stars&per_page=100&page=1
```

Nous avons aussi ajouté les 27 projets issus de la liste *Awesome README*[1], une liste de projets GitHub contenant une «belle» (*beautiful*) documentation README révisée par 33 développeurs. Cette liste contient aussi une description de la qualité de la documentation, comme la clarté des intentions et explications, l'utilisation d'exemples et d'abstractions.

Nous avons donc formé notre corpus en utilisant les listes de projets suivantes :

- Les 27 projets issus de la liste *Awesome README* ;
- Les 100 premiers projets avec plus de 10 000 étoiles (*most-starred*) ;
- Les 100 premiers projets possédant entre 5 000 et 9 999 étoiles ;
- Les 100 premiers projets possédant entre 1 000 et 4 999 étoiles ;
- Les 100 premiers projets possédant entre 100 et 999 étoiles ;
- Les 100 premiers projets possédant entre 10 et 99 étoiles.

En excluant les doublons, nous avons obtenu un total de 517 projets à analyser. Un classification manuelle de ces projets en fonction de leur description a permis de les regrouper en quatre groupes :

1. `https://github.com/matiassingers/awesome-readme`

237 bibliothèques destinées à être importées puis utilisées ou étendues ;

129 applications destinées à être exécutées par l'intermédiaire d'une interface graphique ou de la ligne de commande ;

8 langages de programmation incluant les outils des chaînes de compilation et, possiblement, les bibliothèques standards ;

143 autres projets ne contenant pas de code, comme des livres électroniques, des liste de programmes, de ressources, des curriculum vitae, ou encore des exercices de programmation ou des dépôts d'exemples, et enfin des projets qui ne sont pas documentés en anglais.

Les bibliothèques et programmes sont deux catégories déjà déterminées par Ikeda *et al.* (2018). Nous avons ajouté aussi la catégorie *langage*, puisque notre corpus contient des dépôts pour les outils de plusieurs langages comme `go`, `TypeScript` et `rust`.

GitHub héberge aussi un grand nombre de dépôts qui ne sont pas des projets logiciels (Kalliamvakou *et al.*, 2014). Nous avons exclu les projets de la catégorie *autre* de notre étude afin de ne conserver que les projets les plus représentatifs d'utilisation des fichiers README, tel que fait dans l'étude de Prana *et al.* (2018).

Au final, nous avons conservé un ensemble de 374 projets regroupant les bibliothèques, les programmes et les langages. La taille de ce corpus est donc comparable à celle de l'étude de Prana *et al.* (2018).

La figure 2.1 donne la répartition des projets en fonction des langages d'implémentation qu'ils utilisent. Les projets JavaScript sont les plus fréquents avec 120 occurrences (32.3% des projets).

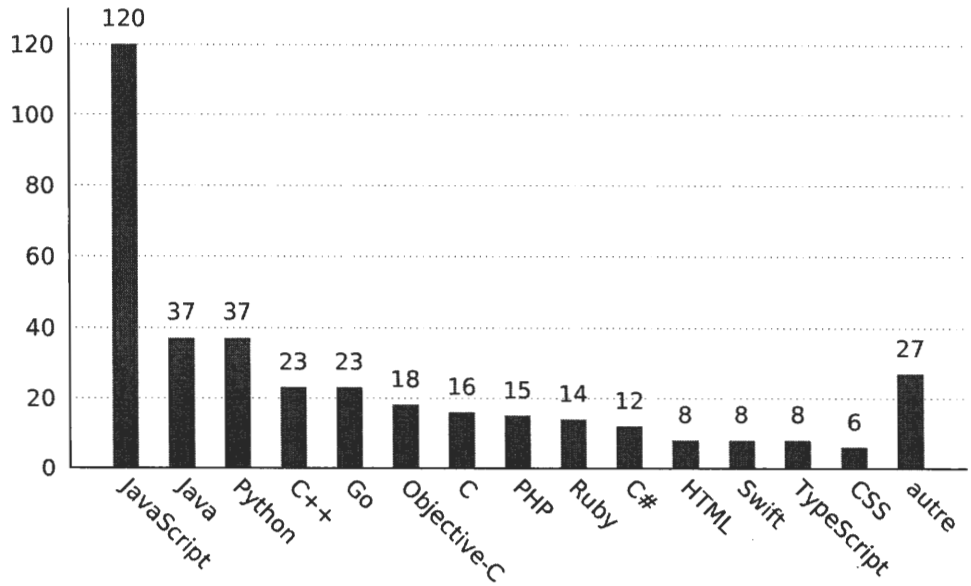


Figure 2.1: Nombre d'occurrences de chaque langage d'implémentation des projets de notre corpus.

2.2 Utilisation du fichier README par rapport aux autres supports

En plus des documents README, GitHub offre quatre autres supports à la documentation des projets :

— L'onglet *issues* permet aux utilisateurs de *poster* des questions aux mainteneurs du projet [2]. Ces questions peuvent correspondre à des remontées de bogues, des questions sur l'installation ou l'utilisation du code, ou encore à des demandes de modification.

— L'onglet *wiki* permet aux mainteneurs du projet de créer un wiki directement dans GitHub. Le wiki est décomposé en pages de documentations organisées en rubriques [3].

2. `https://guides.github.com/features/issues/`

3. `https://guides.github.com/features/wikis/`

— L'onglet *pages* permet aux mainteneurs de créer des pages de contenu statique. Les pages sont souvent utilisées pour présenter de la documentation, des cas d'utilisations ou pour simuler des blogs discutant du projet [4].
— Enfin, la zone de texte *homepage* permet aux mainteneurs de préciser un lien vers le site officiel du projet.

Nous présentons l'utilisation de ces supports de documentation par les projets de notre corpus à la figure 2.2. Les données ont été collectées grâce à l'API REST de GitHub [5].

Sur les 374 projets sélectionnés, seuls quatre (4) ne contiennent aucun fichier README, donc 99% des projets proposent une documentation au format README. Les *issues* sont elles aussi populaires dans les projets avec 87% des projets les utilisant. Les *wikis* et *homepage* sont présents dans respectivement 64% et 60% des projets, donc moins utilisés que les fichiers README ou les *issues*. Finalement, les *pages* sont le support le moins populaire et utilisées dans seulement 32% des projets.

À la question « Est-ce que les projets utilisent effectivement des fichiers README ? », nous pouvons donc affirmer que oui, les projets fournissent presque tous un fichier README. Le README est même le support préféré des écrivains par rapport aux autres alternatives de documentation proposées par GitHub.

2.3 Format des fichiers README

Les plate-formes de code supportent plusieurs formats pour la rédaction des fichiers README. Par exemple, GitHub supporte une liste de neuf formats dont Markdown (Gruber, 2004), reStructuredText (Goodger, 2016) et AsciiDoc (Rackham,

4. `https://guides.github.com/features/pages/`

5. `https://developer.github.com/v3/?`

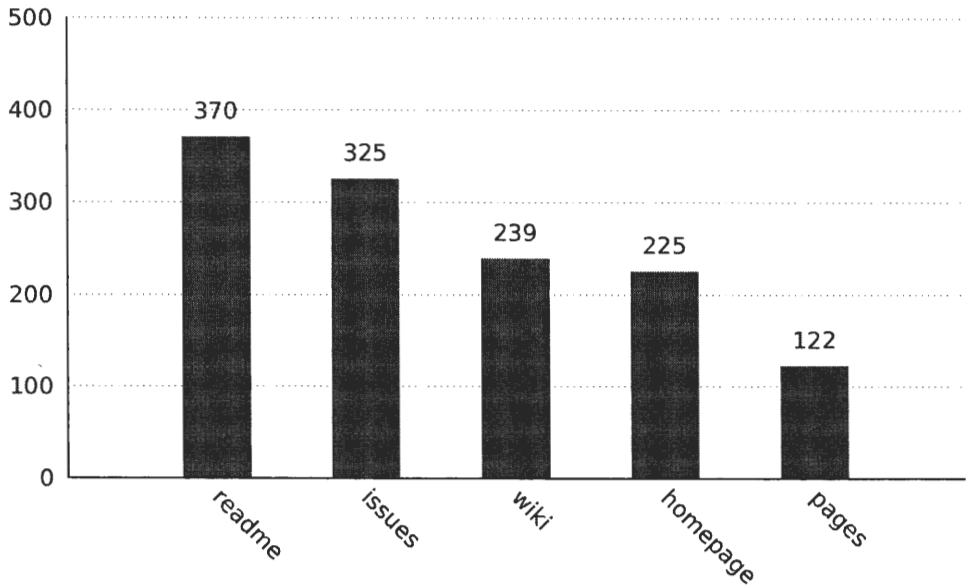


Figure 2.2: Supports de documentation utilisés par les projets de notre corpus.

2002). Nous nous sommes demandé quel était le format le plus populaire parmi notre corpus de projets.

Le format le plus utilisé est clairement le Markdown, utilisé dans 334 projets. Seuls 36 projets utilisent un autre format dont :

- 24 projets avec reStructuredText ;
- 5 projets avec AsciiDoc ;
- 4 projets au format texte brut ;
- 3 projets dans un format inconnu.

Markdown est un langage de balisage léger offrant une syntaxe facile à lire et à écrire (Gruber, 2004). Pour un lecteur, ce langage a l'avantage d'être compréhensible dans son format brut, et ce sans limiter la lisibilité du document comme c'est le cas avec d'autres langages de balisage tels que XML ou HTML. Pour un écrivain, le langage Markdown est facile à utiliser car basé sur une syntaxe légère, comprenant peu de constructions syntaxiques à utiliser pour la mise en forme du document.

Enfin, Markdown peut être facilement compilé vers d'autres langages de balisage de texte comme HTML ou LaTeX. Dans les sections qui suivent, nous introduirons les principaux éléments de Markdown lorsque nécessaire pour en souligner les utilisations typiques rencontrées dans les fichiers README.

Dans la grande majorité des cas — 301 sur 334 —, le fichier README est nommé `README.md`. Nous avons aussi observé quelques variations telles que :

— `readme.md` — 22 projets ;
— `README.markdown` — 6 projets ;
— autres variations autour de la casse — 8 projets.

À la question « quel est le format le plus utilisé pour rédiger les fichiers README ? », la réponse est donc le Markdown dans la plupart des cas. En fait, Markdown est le format par défaut de rédaction des README, des questions (*issues*) et des requêtes d'intégration (*pull-requests*) sur les plate-formes GitHub (Github, Inc., 2012a) et GitLab ce qui peut expliquer nos observations ainsi que celles de Prana *et al.* (2018).

2.4 Taille des fichiers README

Les projets avec plus d'étoiles sont généralement de plus grande taille — *en nombre d'octets*. La figure 2.3 présente la taille moyenne des projets en fonction du nombre d'étoiles. On peut remarquer que les projets de la liste *awesome* sont plus petits que les autres, avec une moyenne de seulement 6 449 octets, soit moins que les projets de 10 à 100 étoiles.

La quantité de documentation, calculée *en nombre de lignes* dans le fichier README, augmente elle aussi avec le nombre d'étoiles. La figure 2.4 présente le nombre moyen de lignes en fonction du nombre d'étoiles. La taille moyenne est de 172 lignes.

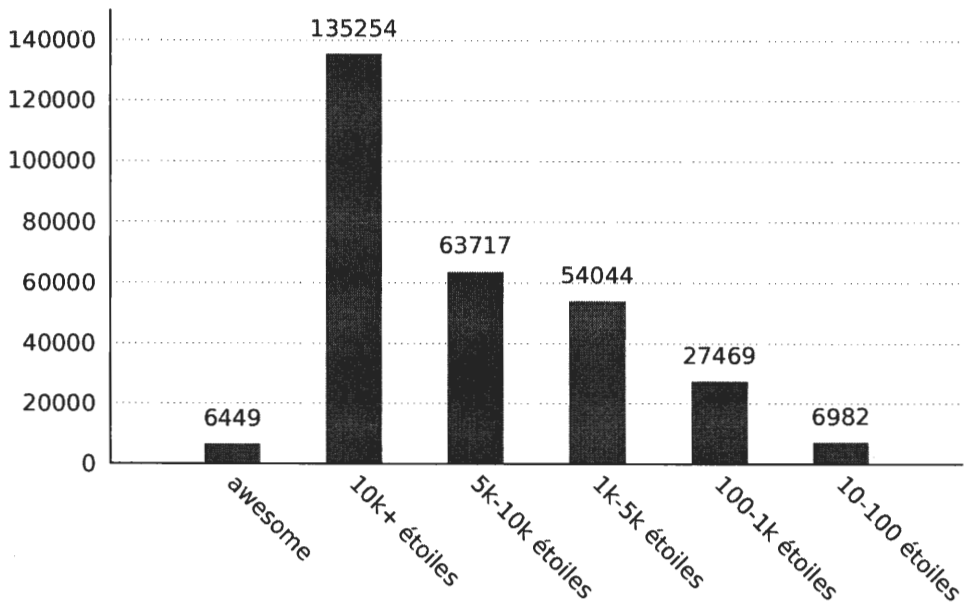


Figure 2.3: Taille des projets (nombre moyen d'octets) en fonction des listes de projets de notre corpus.

En général, on peut donc remarquer que plus un projet a d'étoiles, plus il est de grande taille et plus son README est long. Pourtant, une différence majeure est à observer pour les projets de la liste *awesome*. En effet, ces projets figurent parmi les plus petits mais leurs fichiers README sont parmi les plus gros. Cette corrélation montre un préférence des auteurs de la liste *awesome* pour des ratios taille documentation / taille du code plus élevés que la moyenne des utilisateurs GitHub.

Un README plus long est-il significativement de bonne qualité ? Selon Lethbridge *et al.* (2003), un document trop long décourage le lecteur et rend l'accès à l'information pertinente plus difficile. Néanmoins, si l'on en croit Robillard et DeLine (2010), une longue documentation n'est pas un obstacle à l'apprentissage des API. En effet, « les développeurs sont habitués à survoler de gros blocs de texte à la recherche de l'information qu'ils désirent dans une documentation suivant une

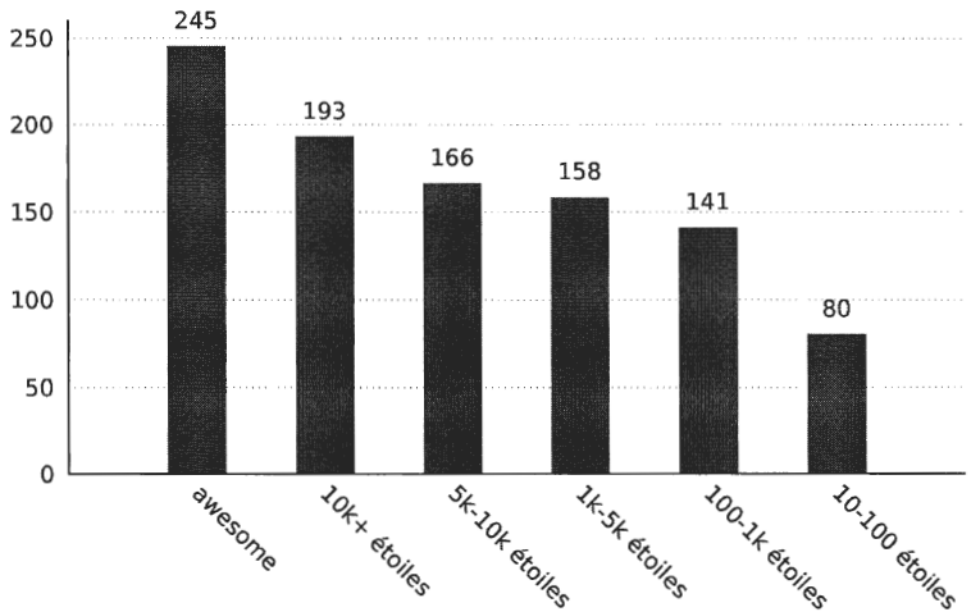


Figure 2.4: Nombre moyen de lignes des fichiers README en fonction des listes de projets de notre corpus.

narration continue ». L'étude de Forward et Lethbridge (2002) montre que la taille de la documentation n'est que le douzième critère considéré par les lecteurs pour en juger sa qualité.

2.5 Structuration des fichiers README

Markdown propose trois constructions permettant de structurer le contenu d'un document, à savoir les en-têtes (*headers*), les paragraphes et les règles. Un en-tête est indiqué en préfixant la ligne par un à six croisillons (#) et en spécifiant un titre. Les en-têtes sont donc utilisés pour structurer un document en sections et sous-sections, le nombre de croisillons d'un en-tête indiquant son niveau dans l'arborescence. Un paragraphe est indiqué par une ou plusieurs lignes de texte ; les paragraphes sont séparés par au minimum une ligne vide. Pour ajouter une

```
# Titre pour en-tête de niveau 1

Ceci est un premier paragraphe.

## Titre pour en-tête de niveau 2

Ceci est un second paragraphe.

Et un troisième. Les paragraphes peuvent être longs
et comprendre plusieurs lignes, comme c'est le cas
pour celui-ci.

***
```

Figure 2.5: Exemple de Markdown avec en-têtes et paragraphes.

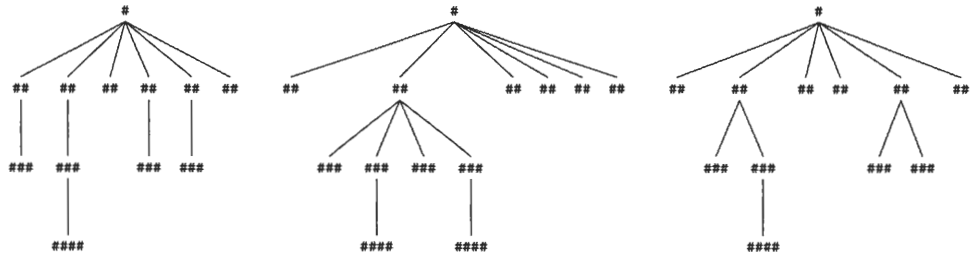
règle (*horizontal rule*), il suffit d'inclure une ligne ne comprenant que trois étoiles successives (***). Voir figure 2.5.

Notre corpus de projets contient un total de 4 286 en-têtes Markdown (de différents niveaux) et 14 173 paragraphes. Le nombre d'en-têtes est proche de celui de l'étude de Prana qui en comprend 4 226. Les règles sont peu utilisées avec seulement 91 occurrences dans notre corpus.

2.5.1 Profondeur des en-têtes

Les niveaux d'en-tête utilisés dans les documents sont donnés en figure 2.8. En moyenne, un document contient 11.5 en-têtes dont 1.3 de niveau 1, 5.7 de niveau 2, 3.8 de niveau 3 et 1.4 de niveau 4.

Les en-têtes de niveaux 2 et 3 représentent 75.3% de notre corpus avec respectivement 1 928 et 1 297 occurrences. Ceux de niveaux 1 et 4 sont moins utilisés avec seulement 445 (10.4%) et 483 (11.5%) occurrences, alors que les niveaux 5 et 6 sont rarement utilisés.



(a) (b) (c)

Figure 2.6: Trois exemples d'arbres correspondant aux caractéristiques moyennes des fichiers README de notre corpus.

Si l'on interprète la structure de la documentation comme une arborescence de sections (niveau 2), sous-sections (niveau 3), etc. indiquées par les en-têtes, dont la racine est l'en-tête de niveau 1, l'arbre moyen des fichiers README de notre corpus est donc surtout «large» au niveau supérieur, peu profond, et peu «touffu». Cette observation ne figure ni dans l'étude de Ikeda *et al.* (2018), ni dans celle de Prana *et al.* (2018). Elle ne semble pas non plus correspondre à la structure attendue d'un document complexe et structuré : en effet, une moyenne de 5.7 en-têtes de niveau 2 pour 3.8 de niveau 3 représente un document avec plusieurs sections, mais peu de sous-sections — et encore moins de sous-sous-sections. En simplifiant, on peut représenter la structure typique d'un arbre ayant ces caractéristiques telle qu'illustré dans la figure 2.6, qui présente quelques arbres avec une unique racine (document principal), six sections, quatre sous-sections et une ou deux sous-sous-sections. Pour comparaison informelle, la figure 2.7 correspond à l'arborescence de la présente thèse.

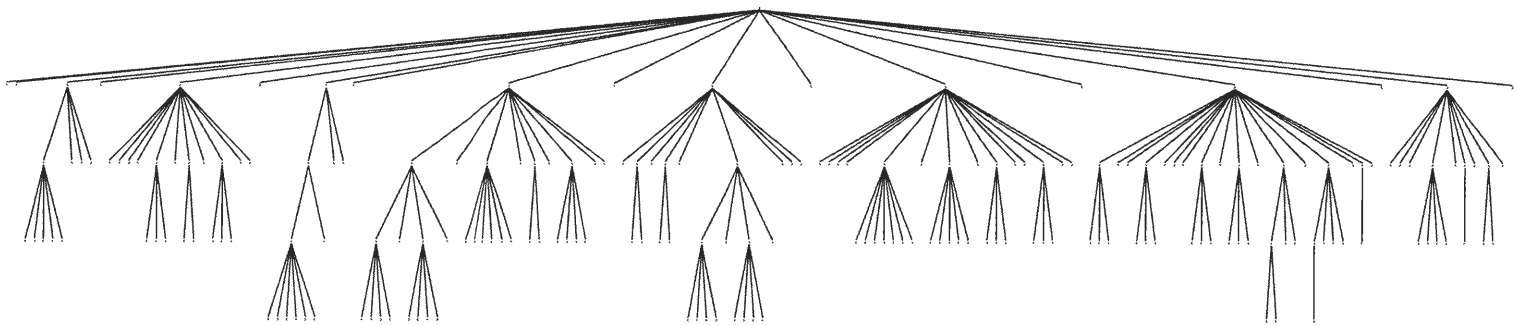


Figure 2.7: Arbre représentant la table des matières de la présente thèse, à l'exception des annexes : la racine est le document dans son ensemble, les enfants de premier niveau sont les chapitres, puis viennent les sections, sous-sections, etc. Seules les (sous-)sections *numérotées* ont été incluses.

L'histogramme présenté en figure 2.9 indique le nombre moyen d'en-têtes de chaque niveau par rapport à la liste d'origine des projets de notre corpus. On remarque que les projets de la liste *awesome* sont généralement plus structurés avec une moyenne de 8.4 en-têtes de niveau 2 et 8.0 de niveau 3.

L'utilisation de la structure est identique entre les projets de plus de 5 000 étoiles et ceux de plus de 10 000. Plus le nombre d'étoiles diminue, plus le nombre d'en-têtes diminue aussi. Cette observation peut être corrélée avec celle sur la taille des documents que nous avons présentée en figure 2.4. Ainsi, plus le document est de grande taille, plus il contient d'en-têtes, donc de sections et sous-sections.

La fréquence d'en-têtes de niveau supérieur à 4 reste faible quelle que soit la liste d'origine du projet. Les en-têtes de niveau 1 sont généralement utilisés une seule fois dans le document quelle que soit la liste d'origine — bien que ce ne soit pas toujours le cas : voir la prochaine section.

2.5.2 En-tête de premier niveau

Sur les 337 projets pourvus d'un fichier README au format Markdown, 266 d'entre eux (79%) contiennent au moins un en-tête de niveau 1 mais seulement 47 (13.9%) contiennent plus d'un en-tête de niveau 1. Un en-tête de niveau 1 est le premier élément Markdown du document dans 224 documents (66.5%).

Une revue manuelle de ces en-têtes de niveau 1 permet d'observer que 261 d'entre eux (77.4%) contiennent le nom du projet auquel le README appartient. Dans 210 cas (62.3%), l'en-tête contient uniquement le nom du projet. Dans 51 cas (15.1%), le nom du projet et accompagné d'une description ou d'un slogan.

Ainsi, les en-têtes de niveau 1 sont utilisés pour présenter le nom du projet, éventuellement avec une description, et sont positionnés au tout début du document, ce qui semble logique. De plus, si on interprète un en-tête de niveau 1 comme

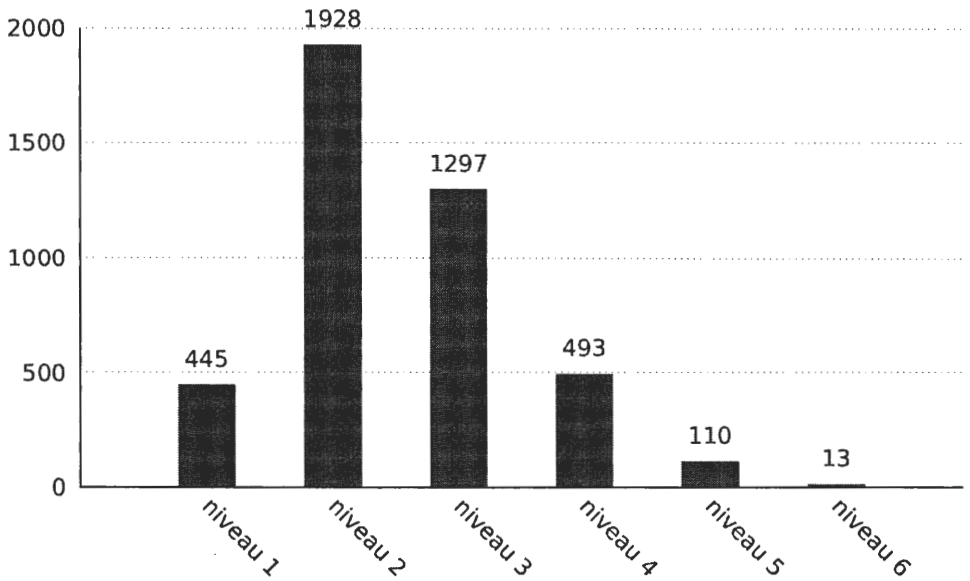


Figure 2.8: Nombre d'occurrences de chaque niveau d'en-tête détecté dans notre corpus.

une indication du titre du document dans son ensemble, il semble alors «naturel» qu'un seul en-tête de ce niveau soit présent.

Le bureau 18F définit une bonne description de projet comme suit : « La description d'un dépôt doit être claire, concise et descriptive. [...] Quiconque visite une page GitHub devrait être capable de dire ce que fait le projet juste en lisant la description » (18F, General Services Administration, 2016b).

Notons que les 337 dépôts analysés possèdent effectivement une description du projet dans la zone de texte prévu à cet effet. Il serait donc facile d'ajouter la description du projet après le titre du projet automatiquement.

L'utilisation d'une description dans le titre ne fait pas pour autant une bonne documentation. Les projets de la liste *awesome* n'ont pas une proportion de description plus élevée que les projets des autres listes.

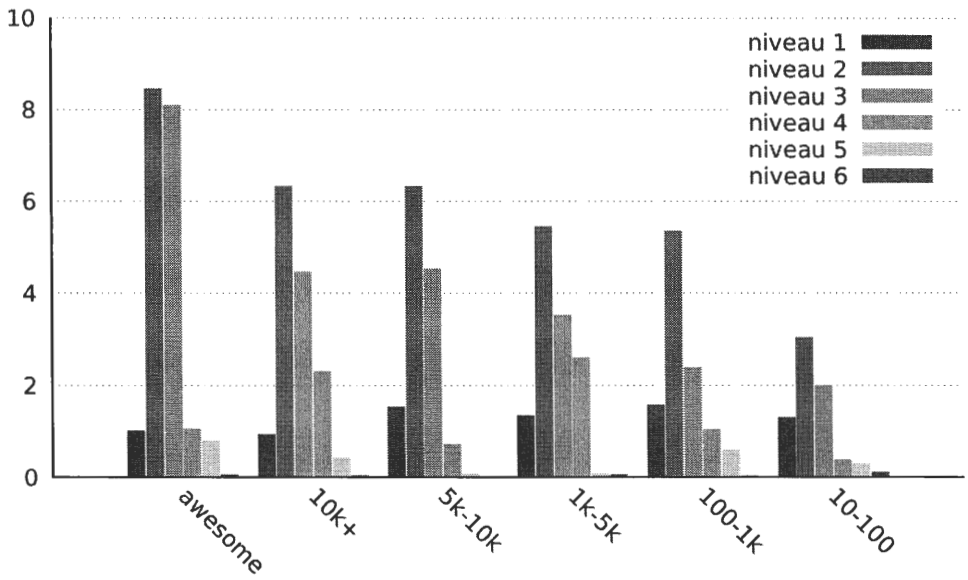


Figure 2.9: Nombre d'occurrences de chaque niveau d'en-tête détecté en fonction des listes de projets de notre corpus.

2.5.3 Catégories des sections

Nous avons aussi analysé les sections contenues dans les fichiers README, telles que spécifiées par les titres indiqués dans les en-têtes, tout niveaux confondus, et nous avons comparé nos résultats avec ceux d'Ikeda *et al.* (2018) et de Prana *et al.* (2018).

Pour la catégorisation des sections, nous avons suivi la même approche qu'Ikeda *et al.* (2018) en réutilisant les sections proposées dans leur article. Les titres des sections sont lemmatisés à l'aide du processeur de langue naturelle StanfordNLP (Manning *et al.*, 2014). Les sections sont ensuite annotées à l'aide d'expressions régulières sur les thèmes, tel que proposé dans l'article. Notre corpus étant plus petit que celui d'Ikeda, nous avons effectué une revue manuelle des annotations.

Tableau 2.1: Fréquence d'apparition des thèmes détectés dans les documents du corpus d'Ikeda *et al.* (2018).

Thème	**Traduction**	**% doc.**	**Description**
Utilisation	*Usage*	60.8	Exemple d'utilisation basique du projet
Installation	*Install*	59.4	Comment installer le projet
Licence	*License*	36.2	Type de licence appliquée au projet
API	*API*	24.3	API du projet
Option	*Option*	23.8	Liste des options du projet
Version	*Release history*	13.3	Historique du projet
Contribuer	*Contribute*	11.2	Comment contribuer au projet
Tests	*Test*	9.9	Commandes de test du projet
TODO	*Todo*	9.8	Liste des TODO du projet
Résumé	*Overview*	9.6	Description du projet
Statut	*Status*	7.9	Statut actuel du projet
Doc.	*Document.*	7.7	Plus de documentation
Auteurs	*Author*	5.9	Liste des auteurs
Aide	*Support*	3.5	Comment obtenir de l'aide
Fonctionnalités	*Features*	3.1	Liste des fonctionnalités du projet
Relié	*Relate.*	2.9	Lien avec d'autres projets
Problèmes	*Issues*	2.4	Problèmes connus et comment les rapporter
Démo.	*Demo*	1.9	Exemple de sortie du projet
Objectif	*Purpose*	1.9	Objectifs du projet
Réfs.	*Refer.*	1.8	Références et remerciements

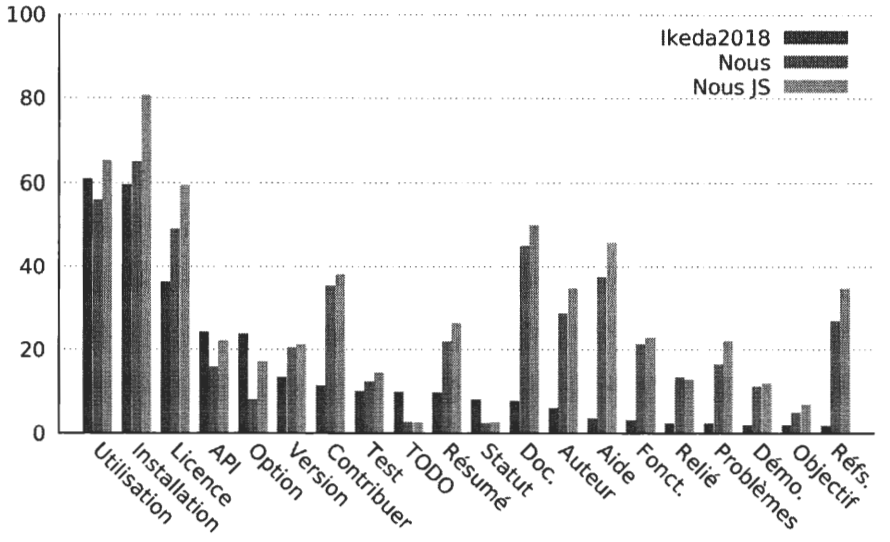


Figure 2.10: Comparaison des fréquences d'apparition des thèmes dans le corpus d'Ikeda *et al.* (2018) (`Ikeda2018`) par rapport aux thèmes trouvés dans notre corpus (`Nous`) et aux projets JavaScript de notre corpus (`Nous JS`).

La liste des thèmes issus de l'article d'Ikeda est présentée dans le tableau 2.1. Les sections sont triées par fréquence d'apparition dans les documents de leur corpus selon un ordre décroissant, du plus fréquent au moins fréquent.

Nous avons comparé nos résultats avec ceux d'Ikeda en figure 2.10. La série *Ikeda2018* correspond au pourcentage de documents de leur corpus contenant chaque thème. Les projets analysés par Ikeda *et al.* (2018) sont tous en JavaScript. La série *Nous* correspond au pourcentage de documents dans notre corpus contenant chaque thème. La série *Nous JS* correspond au pourcentage de documents relatifs à des projets JavaScript dans notre corpus contenant chaque thème.

Les résultats sont comparables pour les thèmes les plus fréquents (partie de gauche de l'histogramme, fig. 2.10). Par contre, notre corpus contient un plus grand nombre de thèmes, et en contient plus souvent (partie droite). Cette différence peut s'expliquer par le protocole de sélection des projets du corpus. En effet, Ikeda

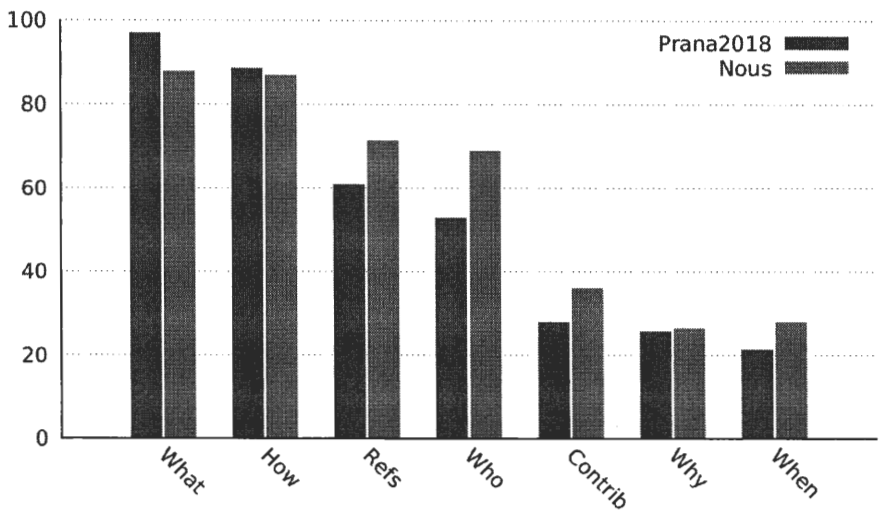


Figure 2.11: Comparaison des fréquences d'apparition des thèmes dans le corpus de Prana *et al.* (2018) (`Prana2018`) par rapport aux thèmes trouvés dans notre corpus (`Nous`).

et al. ont choisi d'analyser 43 000 projets JavaScript sans considération sur la taille du README ou du statut du projet. Nous nous sommes concentré sur 337 projets uniquement, tous des projets comptant au minimum 10 étoiles. Ainsi, il est possible que nos projets contiennent en moyenne plus d'informations que les projets du corpus d'Ikeda *et al.*.

Comparer les résultats d'Ikeda *et al.* (2018) avec ceux de Prana *et al.* (2018) est une tâche complexe. En effet, nous ne possédons qu'une quantité limitée d'information sur la composition du corpus utilisé par Prana *et al.*, et les thèmes de sections qu'ils utilisent sont transversaux à ceux choisis par Ikeda *et al.*. Néanmoins, nous pouvons comparer nos résultats à ceux de Prana *et al.* en réutilisant leur taxonomie. La figure 2.11 donne le pourcentage de documentation issus de leur corpus contenant chaque thème en comparaison avec notre corpus. Les résultats sont similaires pour chaque thème.

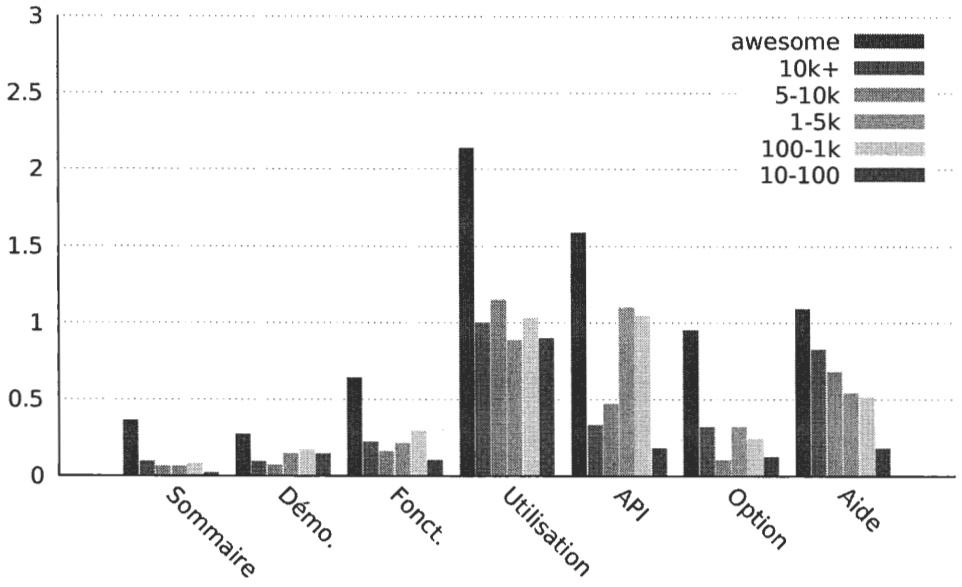


Figure 2.12: Nombre moyen de sections par thème en fonction des listes de projets de notre corpus.

Enfin, nous avons comparé l'utilisation de ces thèmes par rapport aux listes d'origine des projets de notre corpus. La figure 2.12 présente les thèmes pour lesquels on observe le plus de différence en fonction des listes d'origine. On remarque que les projets issus de la liste *awesome* proposent en général plus de sommaires, d'explications sur les fonctionnalités, de démonstrations, de documentation d'utilisation, de liens avec l'API, de documentation sur les options et d'informations de support.

2.5.4 Ordre des sections

Nous avons aussi comparé l'ordre d'apparition des thèmes dans les README. Pour cela, nous avons noté la position de chaque section trouvée dans un README. Pour limiter le biais dû à la taille du README, nous avons noté la position en pourcentage de la taille du document. Ainsi, une section trouvée au début du document aura un plus faible pourcentage qu'une section trouvée à la fin.

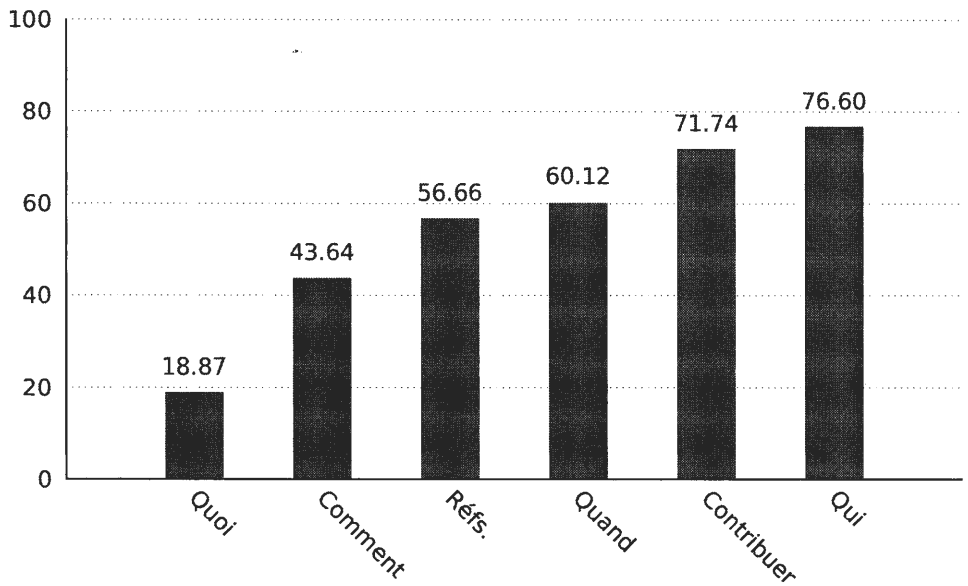


Figure 2.13: Position moyenne des thèmes isolés par Prana *et al.* (2018). La position moyenne est donnée en pourcentage de la taille du document.

La figure 2.13 présente l'ordre des thèmes isolés par Prana *et al.* (2018) en fonction de leur position moyenne dans les documents. Comme on peut le constater, en général un README commence une explication du quoi, suivi du comment puis de références vers la documentation et le code. Viennent ensuite les informations sur la version, comment contribuer et qui contacter en cas de problème.

La taxonomie utilisée par Prana *et al.* est difficile à relier à la réalité des fichiers README. En effet, la division des sections en *quoi*, *comment*, *qui*, *quand* est transversale aux sections réellement utilisées dans les fichiers. Par exemple, le *qui* regroupe à la fois la liste des contributeurs, les informations de support, le code de conduite des contributeurs et la licence. Cependant, selon les recommandations du bureau 18F et de GitHub, ces informations ne sont pas rangées dans les mêmes sections.

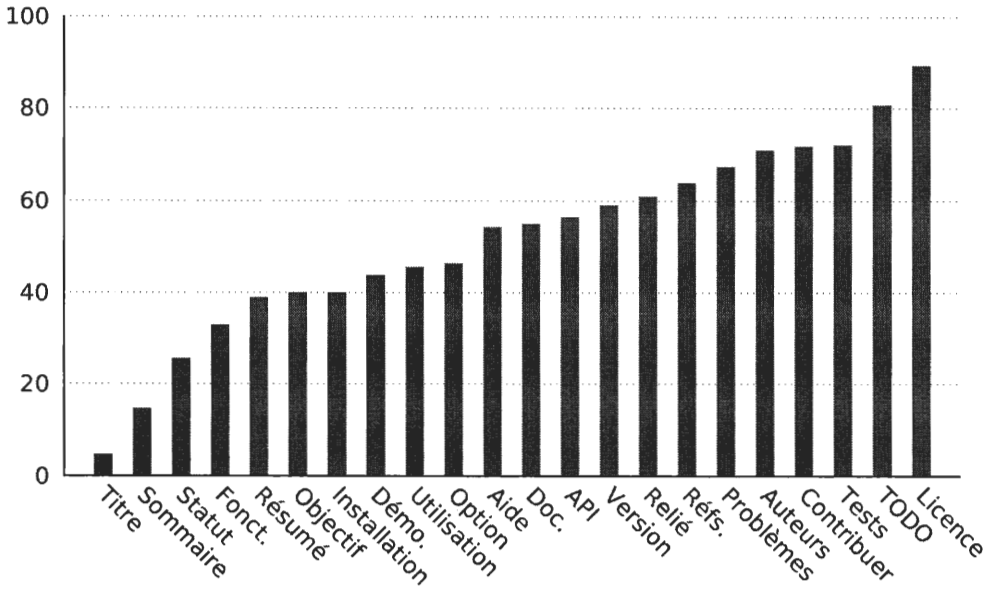


Figure 2.14: Position moyenne des thèmes isolés par Ikeda *et al.* (2018). La position moyenne est donnée en pourcentage de la taille du document.

Nous avons donc utilisé la taxonomie d'Ikeda *et al.* pour déterminer l'ordre des thème dans les fichiers README. La figure 2.14 donne l'ordre des thèmes isolés par Ikeda *et al.* (2018) en fonction de leur position moyenne dans les documents. Cette représentation donne une meilleure idée de l'ordre des sections dans un README. Mise à part la position de la table des matières qui se trouve généralement avant les objectifs du projet dans nos observations, cet ordre est compatible avec celui préconisé par la documentation GitHub. L'ordre observé des sections ne varie pas en fonction du type de projet ou de sa liste d'origine dans le corpus.

2.6 Utilisation de blocs Markdown

Après les sections, nous nous sommes intéressé aux utilisations des *blocs Markdown*. Ces blocs regroupent les blocs de citation, les listes et les blocs de code et permettent eux aussi de structurer le contenu du document.

2.6.1 Utilisation de blocs de citation

Les blocs de citation permettent de relier le contenu du README à d'autres ressources comme des extraits de livres, d'articles ou des propos. Markdown utilise une syntaxe proche de celle des réponses de courriels pour la rédaction de citations :

```
Ceci est un paragraphe sans mise en forme.

> Ceci est une citation.
>
> Ceci est un second paragraphe de citation.

Ceci est un autre paragraphe sans mise en forme.
```

L'utilisation des blocs de citation n'est pas fréquente dans notre corpus, avec seulement 116 occurrences soit 0.3 bloc de citation par document. Les écrivains utilisent souvent le bloc de citation pour accentuer une partie du texte, comme une note importante sur les fonctionnalités (32.8% des occurrences) ou le statut du projet (31.9%). Cette utilisation peut s'expliquer par le rendu graphique des blocs de citation par les plate-formes comme GitHub.

Les autres utilisations correspondent à des citations (11.2%), la présentation des procédures d'installations (3.4%) et de la licence (2.5%). Ainsi que d'autres utilisation que nous n'avons pas réussi à catégoriser (18.1%). L'interprétation détaillée des résultats d'analyse sur les blocs de citation est disponible en annexe A.1.1.

2.6.2 Utilisation de blocs de listes

Comme nous l'avons vu, une des caractéristiques d'une bonne documentation est la présence d'abstractions. Les listes et énumérations sont un type d'abstraction et de structuration du contenu. Grâce à l'imbrication des listes, il est possible de représenter des structures arborescentes comme la hiérarchie d'un système de

fichiers ou l'imbrication du code source. Markdown fournit deux types de listes : non-ordonnées et ordonnées.

Les éléments des listes non-ordonnées sont préfixés par les caractères étoile (*), plus (+) ou moins (-) :

```
* pommes
* poires
* bananes
```

Les éléments des listes ordonnées sont préfixés par une puce numérotée :

```
1. casser les oeufs
2. battre les oeufs
3. faire cuire l'omelette
```

Les listes non-ordonnées sont les plus utilisées avec 1 087 occurrences contre 120 listes ordonnées. Le détail de l'analyse des blocs de listes est donné en annexe A.1.2.

Les listes ordonnées sont utilisées pour composer le sommaire du document (33.8% des occurrences) et les étapes d'installation (21.7%) et de lancement du projet (7.4%).

Les écrivains préfèrent les listes non-ordonnées pour donner un aperçu des fonctionnalités des projets (10.9%) et pour faire références aux éléments du code comme les packages, classes ou méthodes (9.5%). Enfin, les listes sont utilisées pour la présentation des auteurs et mainteneurs des projets (5.3%), pour la présentation des liens ou citations de ressources externes (4.3%), ou encore pour lister les changements opérés entre deux versions (5.0%) et autres (2.0%).

2.6.3 Utilisation de blocs de code

Les bonnes pratiques de la documentation recommandent l'utilisation d'exemples au sein de la documentation pour expliquer comment utiliser ou étendre le code

du projet. Markdown permet de présenter un bloc de code en préfixant chaque ligne soit par quatre (4) espaces, soit par un caractère de tabulation.

```
Le bloc suivant est un exemple de code Nit:

    fun hello do
        print "Hello, World!"
    end
```

La plate-forme GitHub propose une extension permettant de formatter des blocs de code en fonction d'un langage spécifique (Github, Inc., 2012b). Ce formatage se matérialise par une coloration syntaxique lorsque le bloc de code est compilé dans un langage de présentation. Afin d'appliquer le formatage, le bloc de code est délimité par des barrières (*fences*) en utilisant trois caractères tilde (~~~) ou trois apostrophes inversées (```). Immédiatement à la suite des caractères de la première barrière, l'écrivain peut préciser le langage dans lequel le bloc de code est écrit.

```
Le bloc suivant est un exemple de code Nit:

~~~nit
fun hello do
    print "Hello, World!"
end
~~~
```

L'analyse détaillée des blocs de code est donnée en annexe A.1.3. Le corpus comprend un total de 2 122 blocs de code ce qui représente une moyenne de 6.3 blocs de code par document. La notation avec barrière est généralement préférée par les écrivains avec 1 745 occurrences (82.2%) contre 377 (17.8%) pour la notation indentée. Cette préférence peut être expliquée par le fait que GitHub colore syntaxiquement le code source si un langage est indiqué. La moyenne est de 5.7 lignes de code par bloc (min. : 1, max. : 149).

Shell est le langage le plus fréquent avec 391 occurrences (22.4%) ; viennent ensuite JavaScript (345 occurrences, 19.7%), puis Java (101 occurrences, 5.7%). Tous

types et langages confondus, les blocs de code sont utilisés pour présenter des exemples d'utilisation de code (41.7%), des instructions d'installation (35.9% des occurrences), d'exécution (18.9%) et autres (3.3%). Fait intéressant, la quasi-totalité des extraits de code identifiés comme du Shell correspondent à des procédures d'installation et d'utilisation (387/391 blocs Shell). Les procédures sont donc généralement données pour des systèmes Unix si l'on compare l'utilisation du langage Shell aux deux (2) seules occurrences du langage Bat.

Si on compare les positions moyennes des différents thèmes de blocs de code, on remarque que les exemples d'installation arrivent en premier (position moyenne à 36% du document). Suivent les exemples d'exécution (43% du document) et, enfin, les exemples d'utilisation (54% du document).

Les projets de la liste *awesome* possèdent le plus de blocs de code relatifs à l'installation. Si l'on compare le nombre d'étoiles, les projets avec le plus d'étoiles sont mieux documentés quant à leurs procédures d'installation.

2.7 Utilisation de constructions Markdown en ligne (*inline*)

En plus des blocs, Markdown propose des constructions simples permettant d'ajouter des liens ou des images et de changer la présentation du texte. Ces constructions ne sont pas associées à des blocs de lignes, c'est-à-dire s'utilisent directement dans le texte (*inline*). Nous avons aussi étudié l'utilisation de ces constructions dans notre corpus.

2.7.1 Utilisation de liens

Les liens permettent de mettre en relation le contenu du fichier README avec d'autres ressources, parfois externes au projet. Markdown supporte la création de documents avec liens. Pour la création d'un lien vers d'autres ressources — les liens vers des images sont décrits à la prochaine sous-section —, le texte du lien est mis entre crochets et son adresse est mise entre parenthèses :

```
Ceci est un lien vers le
[site officiel du langage Nit](http://nitlanguage.org)
```

Le corpus contient 7 776 utilisations de liens dans le Markdown, soit une moyenne de 23.1 par document. Ils sont utilisés pour pointer vers les ressources de documentation (16.6%), les éléments de l'API (16.5%), le site officiel de l'outil ou de la bibliothèque (10.5%), pour composer les sommaires (9.5% des occurrences), des dépôts sur GitHub pour présenter les dépendances d'un projet (8.1%) et le téléchargement de l'outil ou la bibliothèque présentée par le dépôt (3.8%). L'analyse détaillée des liens et leurs autres utilisations est fournie en annexe A.2.1.

2.7.2 Utilisation d'images

Les images représentent un autre type d'abstraction que l'on peut s'attendre à trouver dans les fichiers README après les listes. Les images utilisent la même syntaxe que les liens mais sont préfixées par un point d'exclamation (!). Le texte entre crochets correspond au titre de l'image, aussi matérialisé par le texte alternatif de l'image lorsque compilé vers un langage de présentation. L'adresse fournie entre parenthèses représente l'adresse à laquelle se trouve la ressource à afficher :

```
![Logo Nit](http://nitlanguage.org/logo.png)
```

Notre corpus comprend 1 335 images dont nous faisons l'analyse détaillée en annexe A.2.2. Elles sont utilisées pour présenter les badges (76.7% des occurrences), les

captures d'écran d'outils (14.8%) et les logos de projets (4.9%). Seulement quatre images correspondent à des diagrammes et toutes sont utilisées pour représenter des processus. Pourtant, la présence d'abstraction telles des diagrammes UML fait partie des caractéristiques d'une bonne documentation.

2.7.3 Utilisation de mises en relief de texte

Afin de mieux faire ressortir certains éléments du texte, il est possible d'agir sur le style du texte lorsque compilé dans un langage de présentation. Markdown supporte deux niveaux pour mettre du texte en relief : léger et fort.

La mise en relief *légère* est balisée par une étoile (*) positionnée avant et après le texte à mettre en relief. Elle se matérialise généralement par un texte *en italique* lorsque compilée dans un langage de présentation, comme HTML.

La mise en relief *forte* est balisée par deux étoiles (**). Elle se matérialise généralement par un texte **en gras**.

```
Ceci est une *mise en relief légère*.
Ceci est une **mise en relief forte**.
```

Avec 1 628 utilisations (moy. : 4.9 par document), les mises en relief fortes sont trois fois plus présentes dans notre corpus que les légères, qui ne comptent que 537 utilisations (moy. : 1.6 par document). Chaque type de mise en relief est analysé en détail dans l'annexe A.2.3.

La mise en relief forte est majoritairement utilisée pour présenter des noms d'auteurs ou de contributeurs comme `**Alexandre Terrasa**` (49.4%), des identifiants issus du code tels les noms de classes ou de méthodes (23.4%) et des mots de la langue anglaise (17.9%). La mise en relief légère est utilisée pour le texte en langue naturelle (75.0), les identifiants issus de l'API (21.8%), les numéros de version (2.0%) et les noms d'auteurs (1.1%).

2.7.4 Utilisation de mises en relief de code

La mise en relief de code en ligne au sein d'un paragraphe est entourée par des apostrophes inverses (`). Le code est présenté selon une police à chasse fixe lorsque compilé dans un langage de présentation.

```
Ceci est un exemple de `print "Hello, World!"` dans un
paragraphe.
```

Nous avons dénombré 7 012 utilisations de la mise en relief de code dans notre corpus, soit une moyenne de 20.8 par document. Le détail de leur analyse est donné en annexe A.2.4.

Les mises en relief de code sont généralement utilisées pour montrer un lien entre la langue naturelle écrite dans le fichier README et des entités de l'API que ce fichier documente (98.5%). Parmi ces utilisations, on peut distinguer : des références au code, correspondant à des identifiants comme `‘MaClasse‘` (56.3%). des extraits de code utilisés pour montrer des instructions ou des expressions plus complexes qu'un simple identifiant (17.9%), des noms de commandes à taper dans le terminal (13.6%) et des noms de fichiers, chemins et URLs (10.7%).

2.8 Menaces à la validité de notre étude

Validité de construction La validité de construction représente le degré de confiance accordé à la méthodologie par rapport aux objectifs de l'étude.

Nous avons repris la même méthodologie que Ikeda *et al.* (2018) et Prana *et al.* (2018) et corrélons nos résultats avec les leurs. Nous croyons donc que la menace à la validité de construction de notre étude est minime.

Validité interne La validité interne d'une étude se réfère à la confiance que l'on peut accorder aux résultats obtenus.

La menace majeure à la validité interne de notre étude est le processus d'annotation des sections et constructions Markdown. En effet, là où les études précédentes reposent sur deux ou trois annotateurs, nous n'en avons utilisé qu'un seul. Le manque de recouvrement entre les annotations placées peut aussi constituer un biais dans nos résultats : il est en effet possible qu'un autre annotateur produise des regroupements différents.

Cependant, nous avons réutilisé la même approche que les études précédentes et avons repris les mêmes thèmes que ceux utilisés par Ikeda *et al.* (2018). De plus, les résultats obtenus sur la quantité de thèmes présents par document peuvent être corrélés avec ceux obtenus précédemment par Ikeda *et al.* (2018) et Prana *et al.* (2018). Donc, cette stratégie nous permet de croire que nos résultats soient valides.

Validité externe La validité externe d'une étude correspond à la possibilité de généraliser les résultats obtenus, dans notre cas, de généraliser à l'ensemble des fichiers README existants.

Notre échantillon d'analyse de 337 projets est proche des 393 utilisés par Prana *et al.* (2018). En outre, nous avons tenté de stratifier notre échantillon, en incorporant des projets avec différents niveaux de notoriété (nombre d'étoiles). Il est cependant possible qu'un autre échantillon, basé sur un autre critère que le nombre d'étoiles, mène à d'autres résultats. Le filtrage des projets de la catégorie *autre* pour ne conserver que les projets de type *langage*, *bibliothèque* et *application* peut aussi impacter les résultats obtenus.

En outre, notre étude se limite aux fichiers de format Markdown. D'autres formats de présentation, tels que ReStructuredText ou AsciiDoc, pourraient produire des résultats différents — bien que Markdown semble clairement le format le plus couramment utilisé. De plus, nous ne pouvons pas non plus généraliser notre approche à d'autres types de documentation que les fichiers README.

2.9 Conclusion

Dans ce chapitre nous avons réalisé une étude empirique du contenu de fichiers README provenant de 517 projets issus de la plate-forme de partage de code GitHub. Cette étude nous permet de répondre aux cinq questions relatives à l'utilisation des fichiers README dans le monde réel que nous avions identifiées dans l'introduction.

Les fichiers README sont présents dans 99% des projets GitHub étudiés. Ils constituent le support de documentation le plus utilisé par ces projets, devant les *issues*, *wiki*, *homepage* et *pages* de documentation. À la question « Est-ce que les projets utilisent effectivement des fichiers README ? », nous pouvons donc répondre que oui.

Nous nous demandions aussi « Quel est le format utilisé pour rédiger ces fichiers ? ». Dans les projets dotés d'un fichier README 90% d'entre eux utilisent le format Markdown, de loin le plus populaire par rapport à reStructuredText, AsciiDoc ou le texte brut.

Pour répondre à la question « Est-ce que ces fichiers remplissent les critères d'une bonne documentation ? », nous reprenons chacune des caractéristiques d'une bonne documentation présentées au chapitre 1 :

- **Disponible** : Le fichier README est livré avec le code source. 99% des projets analysés en contiennent un ;
- **Contenu pertinent** : 70% des fichiers README analysés référencent les identifiants issus du code de l'API qu'ils documentent pour une moyenne de 21 références par document.
- **Structuré** : Le README suit une narration continue articulée en sections, sous-sections et paragraphes avec une moyenne de 11.5 en-têtes de sections par document et 43 paragraphes par document ;

— **Contient des exemples** : Les exemples sont présents dans 44% des documents étudiés les documents contiennent en moyenne 2 exemples d'installation , 1 exemple d'exécution et 3 exemples de code ;
— **Contient des abstractions** : L'abstraction la plus utilisée est la liste (ordonnée ou non). Les diagrammes UML et autres représentations visuelles ne sont presque jamais utilisés avec seulement 4 occurrences dans tout le corpus.

« Comment le format utilisé aide à remplir ces critères ? » Le format Markdown propose une syntaxe simple, facile à lire et facile à écrire qui permet à la fois de structurer le document à l'aide d'en-têtes de sections et de paragraphes, de présenter des exemples grâce aux blocs de code, des abstractions grâce aux images et aux listes et des références vers le code grâce aux mises en relief de code. L'utilisation des liens permet de lier le contenu du fichier README aux autres ressources de documentation et artefacts du projet.

Cependant nous pouvons remarquer que les abstractions visuelles sont peu utilisées dans la pratique. Forward et Lethbridge (2002) notent que ces abstractions sont le plus souvent produites grâce à des outils externes à la documentation tels que ArgoUML ou Microsoft Visio. Ce manque peut alors s'expliquer par la difficulté de produire les abstractions visuelles puis de les maintenir synchronisées avec la base de code.

« Quelles sont les pratiques couramment utilisées pour composer un README ? » :

— L'arborescence des sections est peu profonde, les en-têtes de niveaux 5 et 6 sont rarement utilisés ; Les documents contiennent un titre reprenant le nom du projet parfois avec une brève description sur son utilité ; Les sections les plus fréquentes concernent l'installation, l'exécution et les options de configuration des projets. La documentation des entités de l'API et leurs exemples d'utilisation. Plus des informations sur le processus de contribution, de remontée de bogues et de licence.

— Les listes ordonnées sont utilisées pour composer les sommaires, les procédures d'installation et d'exécution. Les listes non-ordonnées sont utilisées pour énumérer les fonctionnalités, les références vers le code des API et les noms d'auteurs.
— Les extraits de code sont présentés à l'aide de code entre barrières qui permet de préciser le langage utilisé dans l'extrait. Les exemples d'installation et d'exécution sont présentés à l'aide du langage Shell. Les exemples de code font généralement entre 1 et 5 lignes.
— Les liens externes au document pointent en majorité vers les ressources de documentation. Les liens internes sont utilisés dans les tables des matières pour naviguer vers les sections du document. Les images permettent de présenter des badges et les logos des projets ;
— Les mises en relief de texte fortes sont utilisées pour présenter des noms d'auteurs ou de contributeurs ; Les mises en relief de texte légères sont utilisées pour souligner des parties du texte ;
— Les mises en relief de code sont surtout utilisées pour faire référence aux entités de l'API.

Finalement, une part importante du contenu des fichiers README analysés correspond à la présentation du code de l'API et de son utilisation. En effet, on retrouve fréquemment des exemples d'utilisation, des listes d'entités issues du code ou encore des références vers des identifiants.

Le fichier README contient une documentation taillée sur mesure, représentant en moyenne quelques centaines de lignes, qui doit être maintenue synchronisée avec le code source. De plus, et en sa qualité de vitrine, le README semble être un artefact de documentation important du projet. Dans le prochain chapitre nous étudions donc les outils existants permettant de supporter le travail de l'écrivain dans sa rédaction des fichiers README.

CHAPITRE III

ASSISTANCE À L'ÉCRITURE DE FICHIERS README ET DE DOCUMENTATION D'API : ÉTAT DE L'ART

Le fichier README représente la vitrine d'un projet ainsi que le premier — et parfois le seul — élément de documentation présenté au lecteur. Il apparaît donc important que son contenu soit bien choisi et synchronisé avec le logiciel qu'il documente.

Lethbridge *et al.* (2003) notent deux difficultés rencontrées lors de l'écriture de documentation, difficultés qui s'appliquent aussi à l'écriture de fichiers README : 1) les écrivains ont du mal à trouver quoi écrire dans la documentation ; et 2) la documentation est souvent périmée.

Puisque toute forme de documentation du code source basée sur une maintenance manuelle est susceptible à la désynchronisation avec le code (Friendly, 1995), dans ce chapitre nous nous intéressons aux outils assistant l'écrivain dans la rédaction des fichiers README. Ces outils doivent 1) suggérer du contenu à l'écrivain ; et 2) maintenir ce contenu *synchronisé* avec celui de l'API.

Dans la section 3.1, nous dressons l'état de l'art des *générateurs de fichiers* README. Ces outils sont conçus spécifiquement pour générer le contenu des fichiers README tels qu'on peut en trouver sur GitHub. Nous montrerons qu'au-delà de la suggestion des sections d'échafaudage, ces outils sont généralement assez limités. Seuls deux outils sur les sept étudiés offrent des fonctionnalités relatives aux sections d'API.

Les *générateurs de documentation d'API*, comme Javadoc, sont des outils produisant un catalogue, souvent au format HTML, des entités de l'API à partir des commentaires issus du code et de directives de documentation. On parle alors d'*auto-documentation*. Les générateurs de documentation d'API sont un support prisés par les développeurs grâce à la confiance que l'on peut accorder à la synchronisation de la documentation avec le code source (Forward et Lethbridge, 2002).

Dans la section 3.2, nous comparons les générateurs de documentation d'API existants et nous identifions la liste des fonctionnalités qui pourraient être empruntées afin de mieux assister l'écrivain dans sa présentation des entités de l'API au sein d'un fichier README.

Enfin, dans la section 3.3 nous établissons la liste des critères d'une bonne solution de création de fichiers README, tirant ses racines à la fois dans les *générateurs de fichiers* README et dans les *générateurs de documentation d'API*.

3.1 État de l'art des générateurs de fichiers README

Les générateurs de fichiers README sont des outils permettant de générer ou d'assister l'écriture des fichiers README. Nous avons sélectionné les outils à comparer à partir de diverses sources :

- La littérature scientifique, en utilisant les moteurs de recherche habituels tels Google Scholar [1] et CiteSeerX [2] ;
- Les dépôts de sources publics, en utilisant les moteurs de recherche de GitHub [3] et GitLab [4] ;

1. `https://scholar.google.ca/`
2. `http://citeseerx.ist.psu.edu/index`
3. `https://github.com/`
4. `https://about.gitlab.com/`

Tableau 3.1: Générateurs de fichiers README étudiés.

Outil	**Langage**	**Référence**
Go ReadMe	Go	Chilton (2017)
Rebecca	Go	Brophy (2017)
generate-readme	JavaScript	Schlinkert (2016)
verb	JavaScript	Pizarro (2015)
Readme Generator	*agnostique*	Dyrynda (2016)
README.md generator	*agnostique*	Pizarro (2017)
grunt-readme-generator	*agnostique*	Howlett (2013)

— Les listes d'outils recommandés pour l'écriture de documentation comme *Beautiful Docs*[5] et *Awesome README*[6].

Nous nous sommes intéressé uniquement aux outils capables de générer des fichiers README que l'on peut ajouter à un dépôt de sources. Ceci exclut donc les générateurs de documentations d'API comme Javadoc ou Sphinx — que nous traiterons à la section 3.2 —, de même que les générateurs de documentations statiques comme ReadTheDocs ou GitBook. Ces derniers produisent un ensemble de fichiers, souvent au format HTML, qui peuvent s'utiliser comme un site à part entière, ce qui dépasse le cadre de la génération d'un fichier README unique que l'on peut inclure avec les sources du projet. Au final, nous avons identifié sept outils de génération de fichiers README, qui sont présentés dans le tableau 3.1.

3.1.1 Dimensions de l'étude

Notre étude s'oriente principalement sur la comparaison des fonctionnalités permettant de 1) produire ; puis 2) maintenir à jour le contenu des fichiers README.

5. `https://github.com/PharkMillups/beautiful-docs`

6. `https://github.com/matiassingers/awesome-readme`

Sur la base de l'étude empirique réalisée au chapitre précédent (chap. 2), nous pouvons répartir les sections composant le contenu des fichiers README en deux catégories :

— Les *sections d'échafaudage* constituent les sections traditionnellement attendues dans un fichier README, telles que le titre, la procédure d'installation ou la licence. Le contenu de ces sections peut être partiellement extrait depuis les méta-données des projets comme le nom, la licence ou les dépendances.

— Les *sections d'API* présentent les entités de l'API, comme les classes ou les méthodes, et expliquent leur fonctionnement grâce à des exemples ou des liens vers la documentation d'API. Le contenu de ces sections peut être partiellement extrait depuis le code source de l'API en cours de documentation.

Pour chacun des outils étudiés nous comparons donc sa capacité à fournir du support à la réalisation des tâches suivantes :

1. Suggérer l'échafaudage du fichier README ;
2. Suggérer les éléments de présentation de l'API ;
3. Ajouter des exemples ;
4. Ajouter des abstractions ;
5. Assurer la synchronisation avec les méta-données du projet et le code source.

3.1.2 Analyse des sept outils sélectionnés

Readme Generator (Dyrynda, 2016) est un outil accessible sur internet grâce au navigateur et assistant la création de fichiers README au format Markdown. Il permet l'échafaudage d'un fichier README selon quatre sections :

1. Titre (*Title*) ;
2. Introduction (*Introduction*) ;
3. Exemples de code (*Highlighted Code Samples*) ;
4. Procédure d'installation (*Installation Instructions*).

Pour chacune de ces sections, l'outil propose un champ texte permettant à l'écrivain d'en saisir le contenu au format Markdown. Chaque champ est accompagné d'une description expliquant ce qui devrait se trouver dans la section liée. Une fois le contenu saisi, l'écrivain peut alors récupérer la source Markdown de son fichier README pour l'ajouter à son projet ou le modifier.

Cette approche a l'avantage de guider l'écrivain quant au contenu et à la structure de son fichier README. En revanche, rien n'est fait pour assister la synchronisation du contenu ou la présentation des exemples et d'abstractions.

generate-readme (Schlinkert, 2016) est un outil permettant d'échafauder un fichier README depuis la ligne de commande pour des projets NodeJS. Une fois lancé via la commande `gen readme`, l'outil affiche des questions dans la console dont les réponses vont permettre de composer le fichier.

Ces questions s'articulent autour des sections traditionnelles d'un README :

1. Nom du projet ? (*Project name ?*)
2. Description ? (*Description ?*)
3. Licence ? (*License ?*)
4. Nom de l'auteur ? (*Author name ?*)
5. Page personnelle de l'auteur ? (*Author URL ?*)

L'outil suggère les réponses aux questions en fonction du contenu du fichier `package.json`, le fichier de description des paquets NodeJS pour la plate-forme

NPM[7]. Pour chaque question, l'écrivain peut utiliser les valeurs par défaut proposées pour l'outil ou les changer.

Une fois toutes les questions répondues, l'outil génère un squelette de fichier README au format Markdown que l'auteur peut ensuite modifier. Les sections proposées dans le squelette sont les suivantes :

1. Titre (*Title*) ;
2. Installation (*Installation*) ;
3. Utilisation (*Usage*) ;
4. Licence (*License*).

La section *Installation* est pré-remplie avec un exemple de commande `npm install`, la commande d'installation des paquets NodeJS, basée sur le nom du projet. La section *Usage* contient un exemple d'importation du package avec une seule ligne : `require(NomDuProjet)`.

Le fichier README ainsi généré peut alors être ajouté au dépôt de sources.

Cette approche présente elle aussi l'avantage de guider l'écrivain quant au contenu et aux sections qui seront présentes dans son fichier README. Toutefois, bien que les informations originales puissent être importées depuis le fichier de description du package, rien n'est fait pour assurer la synchronisation du contenu du README par rapport à ces données. Ainsi, si le fichier de description est modifié, aucune alerte n'est signalée. De plus, aucune aide n'est fournie pour la présentation des exemples ou des abstractions.

Go ReadMe (Chilton, 2017) est un outil accessible sur internet assistant l'écriture de fichiers README pour les bibliothèques du langage Go. Il propose à l'écrivain de remplir des champs de texte qui guident la génération du fichier final.

---

7. `http://npmjs.com/`

Les champs de texte sont les suivants :

1. Projet / Dépôt (*Project / Repo*) :
   — Nom du projet (*Project name*) ;
   — Description brève (*Short summary*) ;
   — Dépôt de sources (*Where is your repo hosted ?*) ;
   — Nom d'utilisateur GitHub (*GitHub username*) ;
2. Description (*Description*) ;
3. Installation / Utilisation (*Install / Usage*) :
   — Procédure d'installation (*Install instructions*) ;
   — Exemple (*Example*) ;
4. Autres sections (*Other headers*) :
   — Contribuer (*Contributing*) ;
   — Contributeurs (*Contributors*) ;
   — Auteur (*Authors*) ;
5. Badges / Licence (*Badges / License*) :
   — Liste de badges à cocher (*GoDoc*, *TravisCI*, *CircleCI*, ...) ;
   — Liste des licences (*MIT*, *FreeBSD*, *Apache 2.0*, ...) ;

À partir des réponses saisies par l'écrivain, l'outil produit un squelette de fichier README au format Markdown que l'écrivain peut alors importer dans son dépôt de sources et compléter.

Le fichier généré contient les sections suivantes :

1. Titre : description (*Title : description*) ;
2. Introduction (*Overview*) ;
3. Installation (*Install*) ;
4. Exemple (*Example*) ;
5. Contribuer (*Contributing*) ;

6. Contributeurs (*Contributors*) ;
7. Auteur (*Author*).

La section *Introduction* est pré-remplie à l'aide des badges sélectionnés par l'écrivain. La section *Installation* contient par défaut la commande d'installation des paquets Go, `go get NomDuProjet`. Les autres sections sont pré-remplies en utilisant les réponses fournies par l'écrivain.

Là encore, cette approche guide l'écrivain quant au contenu et aux sections qui devraient se trouver dans son fichier README, mais aucune fonctionnalité n'est proposée pour l'aider à tenir ce contenu à jour ou à insérer des exemples et des abstractions.

README.md Generator (Pizarro, 2017) est une application Python permettant de générer un fichier README à partir d'un fichier de configuration et d'un fichier de contenu texte.

Avec cette approche, l'écrivain crée d'abord un fichier de configuration nommé `readme.json` dans lequel il peut saisir les informations relatives à son projet :

1. Nom du projet (clé de configuration `PROJECT.NAME`) ;
2. Icône du projet (`PROJECT.ICON`) ;
3. URL du projet (`PROJECT.URL`) ;
4. Auteur du projet (`AUTHOR`) ;
5. Description du projet (`DESCRIPTION`) ;
6. Badges (`BADGES`).

L'écrivain écrit ensuite le contenu du fichier README dans un second fichier nommé `readme.content.md`. Ici, le contenu est libre et l'écrivain peut y créer les sections qu'il désire en suivant la syntaxe Markdown.

Une fois les deux fichiers créés, il suffit de générer le fichier README final à l'aide de l'application Python. Le fichier produit comprend alors un en-tête généré à l'aide des informations du fichier de configuration suivi du contenu du fichier libre. À chaque nouvelle génération, le fichier README produit est mis à jour en fonction des informations de configuration et du contenu libre.

Cette approche permet un premier pas vers la synchronisation du fichier README puisque son en-tête est extrait depuis les informations de configuration. En revanche, aucune aide n'est fournie à l'écrivain pour le contenu libre, les sections, les exemples ou les abstractions.

verb (Pizarro, 2015) est un compilateur de fichier README utilisable au travers d'une API JavaScript/Node. L'écrivain saisit le contenu de son fichier README dans le fichier de gabarit `.verb.md`, lequel est écrit en syntaxe Markdown combinée avec des directives de documentation, qui seront remplacées lors de la compilation du fichier README de sortie.

Les directives sont écrites en suivant la syntaxe `<%= commande %>`, où une directive peut correspondre à une variable ou une fonction assistante (*helpers*). Les directives sont remplacées lors de la compilation vers le fichier README à l'aide de l'API verb. L'utilisateur précise à l'aide de NodeJS quelles sont les valeurs des variables à remplacer et quelles sont les fonctions assistantes à utiliser.

Les directives utilisant deux signes de pourcentage comme `<%%= name %%>` permettent d'inclure le contenu de la clé `name` depuis le fichier de description du package NPM `package.json`.

verb permet aussi de générer une table des matières pour le document à compiler en ajoutant la directive `toc` en commentaire : `<!-- toc -->`, directive qui sera remplacée par la table des matières.

Par défaut, verb propose les fonctions assistantes suivantes :

— `page` : injecte le contenu d'un fichier Markdown dans le README tout en ajoutant ses sections à la table des matières.
— `apidoc` : injecte la liste des propriétés d'un objet (fonctions et variables deéclarées dans le prototype de l'objet) depuis le code.
— `badge` : ajoute l'image d'un badge GitHub.
— `license` : insére le contenu du fichier de licence renseigné dans le fichier de documentation du package `package.json`.

D'autres fonctions peuvent être créées par l'écrivain en fonction de ses besoins.

Cette approche est particulièrement intéressante car elle permet à l'écrivain de maintenir son fichier README à jour avec le code, puisque les directives sont évaluées à chaque compilation. En revanche, verb ne propose aucune fonctionnalité pour aider l'écrivain à choisir le contenu de sa documentation, les sections, les exemples ou les abstractions.

grunt-readme-generator (Howlett, 2013) est un plugin pour le gestionnaire de tâches automatisé Grunt [8] permettant d'assembler un fichier README à partir de documents existants et d'options de configuration.

L'outil requiert le nom du projet et le nom d'utilisateur GitHub de son propriétaire ainsi qu'une liste de fichiers à assembler. Les fichiers sont listés grâce à un fichier de configuration JSON permettant d'associer un chemin sur le disque à un titre de section.

Lorsque lancé via la commande `grunt`, le plugin combine les fichiers spécifiés en un seul fichier README. Chaque fichier est précédé d'une section Markdown dont le titre est extrait depuis le fichier de configuration. L'ordre des sections est déterminé par l'ordre d'apparition des fichiers dans la configuration.

---

8. `https://gruntjs.com/`

L'option `table_of_contents`, lorsqu'activée, ajoute une table des matières sous la forme d'une liste de liens internes au README. Le tableau de configuration `toc_extra_links` permet aussi à l'écrivain d'ajouter des entrées personnalisées à la table des matières. Quant à l'option `banner`, elle permet de spécifier un fichier contenant la bannière ou le logo du projet. Il est aussi possible de spécifier un contenu ASCII-art [9].

Ici, aucune aide n'est fournie à l'écrivain quant aux sections et à leur contenu. Seul le fichier d'exemple de configuration donne une idée des sections à utiliser :

1. Installation (*Installation*) ;
2. Utilisation (*Usage*) ;
3. Options (*Options*) ;
4. Exemple (*Example*) ;
5. Exemple de sortie (*Example Output*) ;
6. Construire et tester (*Building and Testing*) ;
7. Licence (*Legal*) ;

Bien que le contenu du fichier README final soit maintenu à jour grâce à la commande Grunt, rien n'est fait pour s'assurer que le contenu des fichiers partiels, quant à eux, soit à jour. Et aucune fonctionnalité n'est proposée pour l'ajout d'exemples ou d'abstractions.

Rebecca (Brophy, 2017) est un outil en ligne de commande permettant de générer des fichiers README pour des bibliothèques Go.

Avec Rebecca, l'écrivain saisit sa documentation dans un gabarit `README.md.tpl` avec la syntaxe Markdown. Ce gabarit est ensuite compilé grâce à la commande `becca -package=NomDuProjet` afin de produire le fichier README.

9. `https://en.wikipedia.org/wiki/ASCII_art`

Dans le gabarit, Rebecca propose l'utilisation de directives de documentation permettant à l'écrivain d'importer le contenu du code depuis la documentation. Trois directives sont ainsi disponibles :

— `doc` : inclut le contenu du commentaire d'une définition dans le code (structure, fonction, variable, ...) ; il est aussi possible de préciser les lignes à importer.
— `code` : inclut le code source d'une entité dans la documentation, ce qui est utile pour inclure des exemples d'utilisation.
— `output` : inclut le résultat attendu par le test unitaire d'un extrait de code.

Les directives sont automatiquement remplacées par leur contenu lors de la génération du README final avec la commande `becca`, facilitant ainsi la synchronisation du fichier avec le code. Si une entité n'est pas trouvée dans le code ou ne contient pas de test unitaire, un avertissement est signalé.

Bien que cette approche n'apporte aucune aide à l'écrivain quant au choix du contenu du fichier README ou de sa structure, la possibilité d'inclure des extraits de commentaires ou de code en fait un outil pratique pour l'ajout d'exemples et leur synchronisation. Malheureusement, aucune fonctionnalité n'est proposée pour faciliter l'ajout d'abstractions.

3.1.3 Conclusion sur les générateurs de fichiers README

Le tableau 3.2 compare les outils selon les dimensions étudiées. Nous pouvons remarquer deux catégories d'outils : 1) ceux qui se concentrent sur l'échafaudage du fichier README dont Go Readme, Readme Generator, generate-readme, grunt-readme-generator et README.md Generator ; et 2) ceux qui se concentrent sur la présentation des entités de l'API comme Verb et Rebecca.

Tableau 3.2: Comparaison des outils de génération de fichiers README.

Légende: La comparaison est établie selon le support de l'outil pour l'échafaudage du contenu (**Échaf.**), la présentation des éléments de l'API (**API**), la présentation d'exemples (**Ex.**) et l'utilisation d'abstractions (**Abs.**)

	Échaf.		API		Ex.		Abs.	
Outils	Suggestion	Synchronisation	Suggestion	Synchronisation	Suggestion	Synchronisation	Suggestion	Synchronisation
Go ReadMe	•							
Readme Generator	•							
generate-readme	•	•						
grunt-readme-generator	•	•						
README.md Generator	•	•						
Rebecca				•		•		
verb				•				•

L'échafaudage se fait généralement par la suggestion des sections traditionnelles du README que l'écrivain doit ensuite remplir. Trois outils, generate-readme, grunt-readme-generator et README.md Generator, permettent de maintenir le contenu des sections d'échafaudage à jour avec les méta-données du projet.

Pour ce qui est de la documentation de l'API, seuls les outils proposant l'utilisation des directives de documentation, comme c'est le cas avec Rebecca et verb, permettent de garder un lien étroit avec le code de l'API. Ces deux outils permettent aussi à l'écrivain d'importer des commentaires, des extraits de code ou des listes d'entités depuis le code source grâce à des directives insérées directement dans le contenu du fichier README à générer.

Concernant la présentation des exemples, seul l'outil Rebecca permet de maintenir les extraits de code synchronisés avec les entités de l'API grâce à ses directives `code` et `output`. Pour les abstractions, seul l'outil verb propose une solution avec sa directive `apidoc` permettant de lister les entités d'un module.

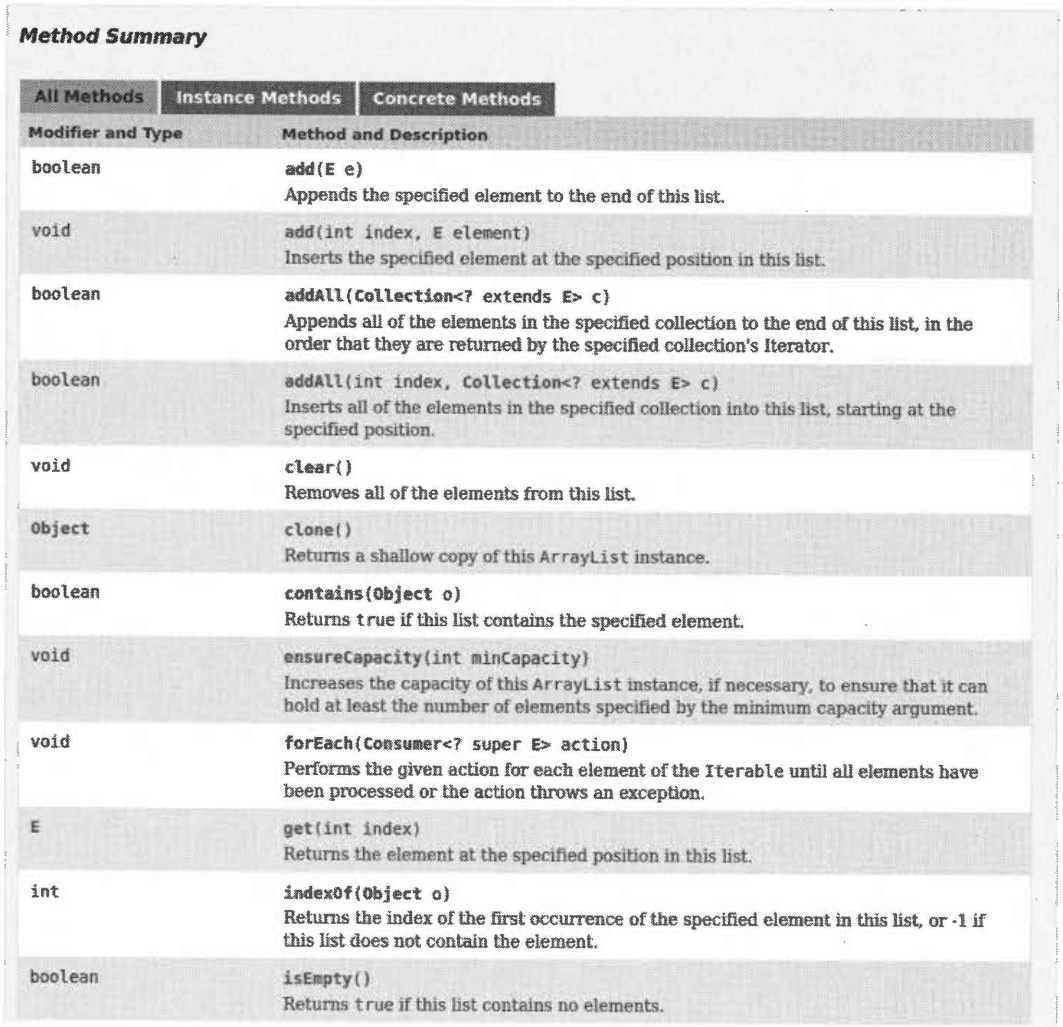
3.2 État de l'art des générateurs de documentation d'API

Le résultat généré par des outils comme Javadoc, RDoc (Thomas, 2001) ou ScalaDoc (Dubochet, 2011) peut être vu comme un catalogue. Ce catalogue contient les éléments du modèle tels les modules, les classes et les méthodes. Le listing 3.1 donne un exemple de code source Java et de commentaire utilisés pour constituer la liste des propriétés de la classe `ArrayList`, alors que la figure 3.1 présente une capture d'écran de la liste résultante produite par Javadoc pour cette classe.

Les listes triées produites par ces outils sont généralement basées sur un ordre lexicographique des identifiants, alors que la catégorisation est basée sur les modules et classes. De tels catalogues sont donc principalement destinés à des utilisateurs *qui savent ce qu'ils cherchent.*

```
/**
 * Resizable-array implementation of the List interface.
 *
 * [...]
**/
class ArrayList<E> extends AbstractList<E> {

  /**
   * Appends the specified element to the end of this list.
  **/
  public boolean add(E e) { [...] }

  /**
   * Inserts the specified element at the specified position
   * in this list.
  **/
  public void add(int index, E e) { [...] }

  /**
   * Appends all of the elements in the specified collection
   * to the end of this list, in the order that they are
   * returned by the specified collection's Iterator.
  **/
  public boolean addAll(Collection<? extends E> c) { [...] }

  [...]
}
```

Listing 3.1: Code Java (partiel) pour la classe `ArrayList` utilisé pour la génération de la documentation d'API avec Javadoc.



Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Appends the specified element to the end of this list.
void	**add(int index, E element)** Inserts the specified element at the specified position in this list.
boolean	**addAll(Collection<? extends E> c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	**addAll(int index, Collection<? extends E> c)** Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear()** Removes all of the elements from this list.
Object	**clone()** Returns a shallow copy of this ArrayList instance.
boolean	**contains(Object o)** Returns true if this list contains the specified element.
void	**ensureCapacity(int minCapacity)** Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	**forEach(Consumer<? super E> action)** Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
E	**get(int index)** Returns the element at the specified position in this list.
int	**indexOf(Object o)** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty()** Returns true if this list contains no elements.

Figure 3.1: Capture d'écran de la liste de propriétés générée par Javadoc pour la classe `ArrayList` du listing 3.1

Bien que la structure de la documentation qu'ils génèrent ne soit pas idéale pour la création de fichiers README, certaines de leurs fonctionnalités peuvent être utiles pour la documentation des entités de l'API dans le README.

Comme nous avons pu le constater avec les outils verb et Rebecca, l'utilisation de directives de documentation pour inclure des éléments traditionnellement présentés par les générateurs de documentation d'API aide les écrivains à maintenir le contenu du fichier README à jour.

Dans cette section, nous dressons un état de l'art des outils de génération de documentation d'API. Nous comparons les outils sur la base des fonctionnalités utiles pour la création d'un README, par exemple, une présentation structurée des éléments de l'API, l'utilisation d'exemples et la création d'abstractions.

Le tableau 3.3 liste les 23 générateurs de documentation d'API que nous avons analysés. La plupart de ces outils sont conçus pour un langage de programmation spécifique, à l'exception de Doxygen, Natural Docs et Universal Report qui supportent plusieurs langages. Nous avons constitué cette liste depuis plusieurs sources, notamment, la page Wikipedia qui liste et compare les fonctionnalités des outils d'auto-documentation [10]. Nous avons aussi utilisé l'index TIOBE [11] afin d'identifier les langages les plus populaires et d'en extraire la liste de leurs générateurs de documentation d'API.

Cependant, nous avons limité notre étude aux outils satisfaisant les conditions suivantes :

1. L'outil est un générateur de documentation d'API – ce qui exclut un outil tel que Pandoc ;

---

10. `http://en.wikipedia.org/w/index.php?title=Comparison_of_documentation_generators`

11. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Tableau 3.3: Générateurs de documentation d'API étudiés.

Outil	**Langage**	**Référence**
Audodoc	Clojure	Faulhaber (2009)
Codox		Reeves (2014)
SandCastle	C#	Microsoft (2006)
Godoc	Go	The Go Authors (2009)
Haddock	Haskell	Marlow (2002)
Javadoc	Java	Friendly (1995)
YuiDoc	JavaScript	Yahoo! Inc. (2011)
JSDoc		JSDoc 3 documentation project (2011)
Doxx		Ribreau (2013)
Docco		Ashkenas (2010)
AppleDoc	Objective-C	Gentle Bytes (2009)
perldoc	Perl	Allen (2014)
Sphinx	Python	Georg Brandl (2007)
PhD DocBook	PHP	HaL Computer Systems (1991)
phpDocumentor		van Riel (2010)
ApiGen		Votruba *et al.* (2010)
Scribble	Racket	Flatt *et al.* (2009)
RDoc	Ruby	Thomas (2001)
YARD		Segal (2007)
Scaladoc	Scala	Dubochet (2011)
Doxygen	*12 langages*	van Heesch (2008)
Natural Docs	*19 langages*	Valure (2003)
Universal Report	*26 langages*	Universal Software (2001)

2. Il cible un langage de programmation – ce qui exclut un outil tel que SQL Documentor ;
3. Il possède un site web fonctionnel, une documentation d'utilisation et est installable et exécutable – ce qui exclut un outil tel qu'Autoduck.

Tous les outils de génération de documentation d'API présentent le contenu de l'API grâce à des listes d'entités accompagnées de leurs commentaires extraits depuis le code. Nous analysons ici chaque outil par rapport aux fonctionnalités supplémentaires qu'il offre et qui pourraient être utiles à la rédaction de fichiers README.

Nous définissons notre analyse selon quatre dimensions principales permettant à l'écrivain d'agir sur les éléments suivants :

3.2.1 La structuration de la documentation de l'API : la sélection, le regroupement et l'ordre de présentation des entités documentées ;

3.2.2 La présentation du code source et des exemples d'utilisation : l'extraction des exemples depuis le code, l'inclusion des exemples dans la documentation et leur synchronisation avec le code ;

3.2.3 L'utilisation des abstractions : la sélection et la présentation des entités documentées dans des abstractions comme des listes ou des diagrammes UML ;

3.2.4 La vérification de la qualité de la documentation produite : l'identification des entités non documentées, mal documentées ou n'existant plus dans le code source.

3.2.1 Structuration

Un manuel utilisateur décrit comment utiliser l'API qu'il documente grâce à des sections organisées pour faciliter l'apprentissage de ses concepts. À la différence d'un catalogue, la structure d'un manuel utilisateur se veut plus flexible et favorise l'expressivité de l'écrivain (Knuth, 1992; Robillard et DeLine, 2010).

Les éléments de l'API peuvent alors être présentés dans un ordre logique, basés sur les cas d'utilisation (*use cases*) et les préoccupations plutôt que dans un ordre fixe lié à la structure du code. Les documentations MSDN (Microsoft, 2004), PerlDoc (Allen, 2014) ou PHP.net (Brown, 2014) sont construites comme des manuels utilisateurs. C'est aussi le cas des fichiers de documentation README.

3.2.1.1 Présentation de la documentation comme un manuel utilisateur

Doxygen propose trois mécanismes pour structurer la documentation d'API générée comme un manuel [12] : (*i*) les *modules*, pour regrouper la documentation de plusieurs entités (fichiers, espaces de noms, classes, méthodes, etc.) dans la même page ; (*ii*) les sous-pages (*subpaging*), pour regrouper les entités dans la même section à l'intérieur d'une page ; (*iii*) les groupes (*member groups*), pour créer des groupes d'entités dans la page de documentation d'une classe.

Haddock permet à l'écrivain de filtrer et trier son catalogue grâce aux listes d'export (*export lists*) [13]. Les entités trouvées dans un module ou un type peuvent être listées manuellement dans les listes d'export afin de changer le contenu du catalogue par défaut. Cette fonctionnalité permet à l'écrivain de choisir le contenu et l'ordre des listes présentées dans la documentation générée. Les groupes nommés (*named*

12. `http://www.stack.nl/~dimitri/doxygen/manual/grouping.html`

13. `http://www.haskell.org/haddock/doc/html/ch03s04.html`

chunks) [14] sont utilisés pour inclure des extraits de documentation textuelle à l'intérieur des listes.

Avec PhD DocBook, l'écrivain peut créer des chapitres au sein de sa documentation afin de créer un livre complet [15]. Les chapitres sont ensuite utilisés pour structurer la documentation et produire un résultat au format HTML décoré avec table des matières et liens entre chapitres.

SandCastle propose la création de *topic files* [16] écrits en XML/MAML pour regrouper les éléments de documentation sous un même sujet (*topic*).

Sphinx fonctionne grâce à l'inclusion de fichiers et les *TOCtrees* [17]. Les pages de documentation peuvent être structurées en incluant plusieurs fichiers de documentation. Pour chaque page, une table des matières est générée à partir des fichiers inclus. phpDocumentor fait la même chose à l'aide du module d'extension (*plugin*) Scrybe [18].

Scribble, l'outil de documentation pour le langage Racket, permet à l'écrivain de documenter ses modules à l'aide de la programmation lettrée (*litterate programming*) (Knuth, 1992). Les commandes Scribble peuvent alors être utilisées pour composer les pages de documentation en important le contenu des modules [19]. Une table des matières peut ensuite être générée à partir du contenu des modules.

---

14. `http://www.haskell.org/haddock/doc/html/ch03s05.html`

15. `http://www.docbook.org/tdg5/en/html/ch02.html#ch02-makebook`

16. `http://www.ewoodruff.us/mamlguide/html/c69a248a-bd94-47b2-90d7-5442e9cb567a.htm`

17. `http://sphinx-doc.org/markup/toctree.html`

18. `http://phpdocumentor.github.io/Scrybe/internals/table_of_contents.html`

19. `https://docs.racket-lang.org/scribble/base.html`

Presque tous les outils supportent l'inclusion de fichiers externes. Cette approche est pratique pour documenter les décisions de conception, les options de configuration ou toute autre information qui n'est pas directement liée au code source. Cependant, ces fichiers ne sont pas synchronisés avec le code source, ce qui ne résout pas le problème de la maintenance.

3.2.1.2 Ordre de lecture

Les suggestions de lecture (*see also*) permettent de suggérer un ordre de lecture dans une documentation au format *catalogue*. En fonction du contexte représenté par la page en cours de lecture, l'outil suggère au lecteur d'autres sujets qui pourraient l'intéresser — par exemple, suggérer la lecture de la documentation pour la méthode `remove` à un lecteur parcourant celle de la méthode `add`.

Plusieurs outils permettent à l'écrivain de spécifier des sujets connexes pour chaque élément de documentation en utilisant des liens. Par exemple, Doxygen utilise l'annotation de documentation `\relatesalso` à insérer dans le commentaire de l'entité à documenter. Les outils APIGen, Javadoc, Scaladoc et YARD suivent la même approche grâce à la directive `@see`, alors que YUIDoc utilise `@cross-links`. Dans SandCastle, les *topics* peuvent être liés à d'autres *topics* reliés grâce à l'utilisation de la balise XML `<seealso>`. Scribble permet à l'écrivain d'ajouter des liens vers la documentation connexes grâce à l'annotation de documentation `@other-doc`.

Cette approche manuelle offre une grande flexibilité à l'écrivain qui peut alors suggérer chaque élément de documentation qui pourrait intéresser le lecteur en fonction de ce qu'il est en train de lire. De plus, comme ces liens sont représentés à l'aide d'annotations, ils peuvent être vérifiés au moment de la génération de la documentation. Ainsi, les outils APIGen et Javadoc peuvent détecter les liens périmés et afficher un avertissement lors de la génération.

En revanche, une telle approche est particulièrement fastidieuse pour l'écrivain qui doit alors insérer les annotations dans chaque élément de documentation ou chaque commentaire. Puisque le code évolue au fil du temps, cette méthode ajoute une charge de travail supplémentaire au mainteneur qui doit ajouter de nouveaux liens dans la documentation existante à chaque fois qu'il modifie le code.

3.2.2 Présentation du code source et d'exemples

La plupart des outils permettent à l'écrivain d'inclure le code source des entités directement dans la documentation. Le code source peut alors être utilisé par le lecteur quand la documentation est insuffisante. Le code fournit aussi une information supplémentaire pour le mainteneur qui peut alors voir comment une entité est implémentée sans avoir à la localiser dans le code source. Pouvoir afficher du code dans la documentation est aussi la première étape pour pouvoir présenter des exemples d'utilisation.

3.2.2.1 Présentation du code

Outre la coloration syntaxique et l'ajout de liens hypertextes permettant la navigation depuis le code source vers la documentation, seuls quatre outils proposent des fonctionnalités utiles pour la compréhension du code par le lecteur.

Godoc utilise le *Go Playground*[20] pour permettre au lecteur de modifier et d'exécuter les extraits de code directement dans le navigateur. Les fonctionnalités du *Go Playground* sont relativement limitées mais il supporte la mise en forme automatique du code et le partage d'extraits grâce à des URL alors que la coloration syntaxique n'est même pas proposée.

---

20. `http://play.golang.org/p/4ND1rp68e5`

Scribble peut importer des modules Racket afin d'en évaluer le code source et d'y ajouter les types statiques lors de la génération de la documentation [21]. Le lecteur peut alors utiliser des commandes Scribble pour interagir avec la console Racket et exécuter le code ligne par ligne.

Inspiré par la programmation explicative (*elucidative programming*) de Vestdam et Nørmark (2004), Docco affiche le code source côte à côte avec la documentation extraite depuis les commentaires et les fichiers de documentation externes [22].

3.2.2.2 Annotation des exemples

Les outils Doxygen, Godoc, JSDoc, SandCastle, ScalaDoc, YARD et YUIDoc proposent une annotation à insérer dans le code source pour spécifier les exemples à présenter dans la documentation. Une fois annotés les exemples sont mis en forme et insérés dans les pages de documentation d'API correspondantes.

3.2.2.3 Vérification des exemples

Trois outils permettent de vérifier les exemples de manière automatique : Godoc, Sphinx et Scribble.

Avec Godoc, les exemples sont écrits dans des fichiers externes qui sont ensuite liés à la documentation grâce à une convention de nommage stricte [23]. Dans le code source de l'exemple, le résultat normalement attendu lors de l'exécution de l'exemple est spécifié en utilisant la directive `"Output"` permettant ainsi de le

---

21. `https://docs.racket-lang.org/scribble/scheme.html`

22. `http://derickbailey.github.io/backbone.marionette/docs/backbone.marionette.html`

23. `https://godoc.org/github.com/fluhus/godoc-tricks#Examples`

vérifier automatiquement. Comme tous les autres extraits de code présentés par Godoc, les exemples peuvent être exécutés directement dans le navigateur grâce au *Go Playground.*

Avec Sphinx, les exemples sont inclus directement dans les commentaires du code et le résultat attendu est décrit grâce à des assertions comme c'est généralement le cas pour les tests unitaires. Un outil externe appelé *Dexy*[24] est utilisé pour les extraire depuis les commentaires, les exécuter et les vérifier.

Dans Scribble, les exemples sont écrits directement dans le code Racket. L'annotation `@examples` est utilisée pour fournir des exemples relatifs à des extraits du code et préciser le résultat attendu pour chaque exemple[25].

Toutes ces approches permettent de maintenir les exemples synchronisés avec le code source et encouragent ainsi les développeurs à fournir des exemples dans la documentation. En effet, en écrivant des exemples, le développeur écrit aussi des tests unitaires pour son code. L'approche proposée par Sphinx fonctionne particulièrement bien avec le développement dirigé par les tests (Beck, 2003) où le développeur alterne fréquemment entre l'écriture de tests unitaires et l'écriture de code.

3.2.3 Utilisation d'abstractions

Listes et arborescences Les listes et les arborescences sont les abstractions les plus utilisées par les générateurs de documentation. Les listes sont utilisées pour regrouper des ensembles d'entités du même type telles les classes ou les méthodes selon un ordre linéaire facilitant ainsi la compréhension du contenu du

---

24. `http://dexy.it/`

25. `https://docs.racket-lang.org/scribble/eval.html`

code. Les arborescences, souvent composées par des listes imbriquées, permettent de représenter la hiérarchie des entités comme par exemple les classes contenues dans un package.

Ces deux représentations offrent une présentation succincte des entités généralement limitée à la première phrase issue de leurs commentaires dans le code accompagnée d'un lien pointant vers leur description complète.

La plupart des outils génèrent une liste de toutes les entités disponibles dans le modèle sans autre filtre que la visibilité (publique et protégée contre privée). Scaladoc et YUIDoc permettent au lecteur de filtrer le contenu des listes selon plusieurs attributs comme le type, l'entité parente ou le nom. Cette fonctionnalité est utile pour un lecteur qui sait ce qu'il recherche mais peut produire beaucoup de bruit pour un néophyte. De plus, cette approche engendre un besoin de micro-gestion et n'est pas persistante lorsque le lecteur ouvre une nouvelle page de documentation.

Selon nous, les listes devraient être triées non pas par le lecteur mais plutôt par l'écrivain, qui sait quelles entités sont utiles en fonction de ce qu'il veut exprimer.

Sphinx fournit un ensemble de filtres qui peuvent être appliqués par l'écrivain pour sélectionner les entités qui doivent apparaître dans les listes en utilisant les directives `automodule`, `autoclass` et `autoexception` dans le code source [26].

Seul Haddock permet à l'écrivain de définir manuellement le contenu des listes d'entités grâce aux listes d'export (*export lists*) [27].

Tous les outils listent les entités suivant un ordre lexicographique des identifiants. ScalaDoc permet au lecteur de changer l'ordre de présentation pour un ordre selon une extension linéaire basée sur les relations d'héritage.

---

26. `http://sphinx-doc.org/ext/autodoc.html#directive-automodule`

27. `http://www.haskell.org/haddock/doc/html/ch03s04.html`

Avec Doxygen, l'écrivain peut désactiver l'ordre automatique pour présenter ses listes selon l'ordre de déclaration dans le code source[28].

3.2.3.1 Diagrammes et figures

Les diagrammes et autres représentations visuelles présentent l'information selon une vue simplifiée et structurée. Ils aident le lecteur à comprendre rapidement le contenu d'une API ou d'un élément de l'API comme un package ou une classe.

Doxygen est bien équipé en ce qui concerne la génération de graphes et de diagrammes[29]. Il permet la génération automatique de diagrammes représentant l'héritage des classes et les dépendances de modules ainsi que des graphes d'appels et des diagrammes de classes UML. Chaque représentation peut être utilisée pour naviguer dans le documentation grâce à des liens hypertextes.

phpDocumentor et ScalaDoc peuvent eux aussi produire des diagrammes de classes. Sphinx et YARD en sont aussi capables grâce à l'utilisation de modules d'extension (*plugins*). Godoc fournit uniquement les diagrammes de dépendances de modules.

Les diagrammes générés automatiquement souffrent des mêmes inconvénients que les listes. En effet, le contenu n'est pas filtré, donc les figures générées peuvent devenir difficiles à lire, voire même difficile à afficher dans le navigateur à cause de leur taille. La documentation utilisateur de l'outil Doxygen donne d'ailleurs un avertissement à ce sujet[30]. Un exemple de diagramme illisible peut être visualisé dans la démonstration en ligne de l'outil phpDocumentor[31].

---

28. `http://www.stack.nl/~dimitri/doxygen/manual/config.html#cfg_sort_member_docs`

29. `http://www.stack.nl/~dimitri/doxygen/manual/diagrams.html`

30. `http://www.stack.nl/~dimitri/doxygen/manual/diagrams.html`

31. `http://demo.phpdoc.org/Responsive/graph_class.html`

Seul Godoc propose un système de filtre sur ses diagrammes de dépendances de modules. Le lecteur peut alors choisir d'afficher ou non les dépendances issues de la bibliothèque standard du langage.

Enfin, Doxygen et Universal Report peuvent présenter le code source sous la forme d'un graphe d'appel navigable[32] mais qui n'est personnalisable ni par le lecteur, ni par l'écrivain.

3.2.4 Vérification de la qualité

Certains outils signalent des avertissements à propos de la documentation manquante lors de la génération du résultat final. Doxygen,[33], SandCastle,[34] et Scribble[35] signalent des avertissements à propos des commentaires manquants sur les entités publiques du code. YARD, à l'aide du module d'extension YardStick[36], permet de signaler des avertissements pour les méthodes publiques ne contenant pas d'exemples. Javadoc, lorsqu'exécuté avec l'option `-Xdoclint`, valide le contenu des commentaires en vérifiant la documentation manquante, la désynchronisation des annotations de documentation avec la signature des méthodes et les erreurs de ponctuation[37]. Scaladoc permet des vérifications semblables grâce au module

---

32. http://www.omegacomputer.com/flowchart.jpg

33. http://www.stack.nl/~dimitri/doxygen/manual/config.html#config_messages

34. https://www.simple-talk.com/dotnet/.net-tools/taming-sandcastle-a-.net-programmers-guide-to-documenting-your-code/#twentyseventh

35. https://docs.racket-lang.org/scribble/core.html

36. https://github.com/dkubb/yardstick/wiki/Rules

37. http://docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html#BEJEFABE

d'extension ScalaStyle[38], de même que Perldoc grâce à PodChecker[39].

Les avertissements sur la documentation manquante offrent un bon rappel à l'écrivain concernant ce qui manque mais peuvent aussi produire beaucoup de bruit par rapport aux entités qui ne nécessitent pas de documentation. Aucun outil ne semble appliquer de vérification plus avancée sur le contenu de la documentation.

3.2.5 Conclusion sur les générateurs de documentation d'API

Le tableau 3.4 résume l'état de l'art que nous avons dressé sur les générateurs de documentation d'API existants.

Sept outils sur les 23 analysés proposent un mécanisme permettant à l'écrivain d'agir sur la structure de la documentation produite, c'est-à-dire, sélectionner les entités à présenter, les regrouper en sections et sous-sections et les ordonner.

Les liens vers d'autres entités de la documentation (*see also*) sont une pratique courante pour établir un sens de lecture. 12 outils utilisent des directives directement dans le code source de l'API pour lister les sujets intéressants pour une entité donnée. Aucun outil ne propose de suggestions relatives à ces listes.

L'inclusion de code source dans la documentation est elle aussi une pratique répandue, et seuls deux outils, AppleDoc et NaturalDocs, ne le permettent pas. L'annotation des exemples grâce à des directives dans le code permet à neuf outils de présenter les exemples directement dans la documentation. Seulement trois outils — Sphinx, Scribble et GoDoc — permettent de vérifier la synchronisation des exemples avec le code source de l'API.

---

38. `https://github.com/scalastyle/scalastyle/wiki/Scalastyle-proposed-rules-(Scaladoc-Comments)`

39. `http://perldoc.perl.org/Pod/Checker.html`

Tableau 3.4: Comparaison de générateurs de documentation d'API selon les fonctionnalités utiles à la génération de fichiers README.

	Structure			Exemples			Abs.		Qualité	
Outils	Sélection	Regroupement	Ordre	Présentation	Annotation	Vérification	Listes	Figures	Style	Synchro.
Doxygen	•	•	•	•	•		•	•	•	•
Sphinx	•	•	•	•	•	•	•			
Scribble	•	•	•	•	•	•				•
ScalaDoc				•	•		•	•	•	•
SandCastle	•	•	•	•	•					•
YARD			•	•	•			•	•	•
Godoc				•	•	•		•		
YUIDoc			•	•	•		•			
Haddock	•	•		•			•			
phpDocumentor	•	•	•	•				•		
Perldoc			•	•					•	•
PhD	•	•	•	•						
Javadoc			•						•	•
ApiGen			•	•						
JSDoc			•	•	•					
Universal Report				•				•		
Docco				•						
RDoc				•						
Doxx				•						
Autodoc				•						
Codox				•						
AppleDoc										
Natural Docs										

Parmi le abstractions utilisées, tous les outils affichent des listes créées à partir des entités de l'API mais seuls cinq outils permettent d'agir sur le contenu de ces listes. Six outils proposent des abstractions visuelles tels des diagrammes UML. Seul GoDoc permet de filtrer le contenu des diagrammes UML et ce filtrage doit être fait par le lecteur.

Par rapport à la vérification de la qualité de la documentation produite, cinq outils permettent de vérifier le style des commentaires utilisés pour documenter les entités et six outils permettent de vérifier les commentaires manquants sur les entités importantes.

3.3 Vers un meilleur support pour la génération des fichiers README

Notre revue des générateurs de fichiers README existants montre que le support principalement offert aux écrivains pour la création de leurs fichiers README se limite à un échafaudage des sections traditionnelles que l'on y trouve. Seuls quelques outils permettent de générer et tenir à jour le contenu de ces sections avec les méta-données du projet.

Les fonctionnalités aidant à documentation des entités de l'API dans le fichier README sont plus rares. Deux outils, verb et Rebecca, proposent l'utilisation de directives de documentation à utiliser dans le fichier README pour y inclure des éléments du code, tels des commentaires ou des extraits de code ou d'exemple.

Les directives de documentation permettent de maintenir un lien étroit avec le code source puisque le contenu à présenter est généré en même temps que la documentation. Ces directives assurent alors une synchronisation parfaite avec le code dès lors que la documentation est regénérée lorsque le code change.

Cependant, aucun outil ne propose à la fois un échafaudage des sections du README et des directives de documentation pour l'API. Et aucun outil n'est capable de

suggérer du contenu à l'écrivain pour la documentation de l'API — les suggestions se limitent aux sections d'échafaudage.

De plus, les directives proposées aux écrivains sont limitées si on les compare aux fonctionnalités couramment implémentées dans les générateurs de documentation d'API. En effet, on retrouve fréquemment la présentation de code, d'exemples ou d'abstractions visuelles dans ces outils.

Nous établissons ainsi les fonctionnalités nécessaires à un bon outil de génération de fichiers README qui inclurait les meilleures fonctionnalités des générateurs de fichiers README actuels et des générateurs de documentation d'API :

— Génération assistée de l'échafaudage du fichier README.
  Les sections traditionnelles des fichiers README comme le titre, la procédure d'installation ou la licence sont générées automatiquement. Leur contenu est extrait depuis les méta-données du projet. Ce contenu est maintenu à jour par l'outil lorsque les méta-données changent.
— Génération des sections dédiées à la documentation de l'API.
  Les sections relatives à la documentation des entités issues du code de l'API sont générées automatiquement. La sélection, le regroupement et l'ordre des entités documentées doivent pouvoir être modifiés par l'écrivain.
— Annotation, vérification et insertion automatique des exemples.
  Les exemples sont sélectionnés depuis la base de code et peuvent être vérifiés automatiquement lorsque la base de code change. Les exemples à utiliser sont suggérés à l'écrivain.
— Insertion automatique des abstractions telles les listes et les figures.
  Le choix des abstractions et de leur contenu doit pouvoir être fait par l'écrivain. Le contenu, le regroupement et l'ordre appliqués aux listes ainsi que le contenu et le niveau de détails des figures doivent être modifiables par l'écrivain. L'outil doit maintenir à jour ces abstractions.

Dans le prochain chapitre (chap. 4), nous nous intéressons à la possibilité de générer automatiquement la totalité du contenu du fichier README, en particulier les sections dédiées à la documentation des entités de l'API. Les chapitres suivants traiterons des outils que nous avons mis en œuvre dans le cadre de cette thèse pour égaler, et même dépasser les outils existants de génération de documentation d'API (chap. 5 et 6) et de génération de fichiers README (chap. 7).

CHAPITRE IV

UNE EXPÉRIMENTATION SUR LA RÉDACTION AUTOMATIQUE DE FICHIERS README

Nous avons montré dans le chapitre 2 que les fichiers README sont organisés autour de sections pré-définies telles l'explication des objectifs du projet, la procédure d'installation et ainsi de suite. Les outils de génération de fichiers README que nous avons étudié au chapitre 3 se concentrent généralement sur l'échafaudage des ces sections.

Cependant, nous avons pu constater que les fichiers README contiennent aussi des sections réservées à l'explication des fonctionnalités de l'API et qu'elles sont directement en lien avec les entités du code source. Dans ce chapitre nous étudions la faisabilité d'une approche permettant de générer ces sections et leur contenu de manière complètement automatisée à partir des entités (modules, classes et propriétés) du modèle d'une application ou d'une bibliothèque.

Nous définissons cette approche de génération automatisée selon trois étapes successives permettant de :

1. Sélectionner les fonctionnalités à présenter dans le fichier README ;
2. Regrouper ces fonctionnalités en sections et sous-sections ;
3. Ordonner le contenu des sections et sous-sections.

Pour chacune de ces étapes, nous cherchons à comprendre la stratégie suivie par un expert quand il écrit sa documentation afin de la reproduire dans une approche automatisée. Pour ce faire, nous avons construit trois expériences impliquant des experts du langage Nit afin d'étudier leurs stratégies à chaque étape de la rédaction de la documentation d'API. Comme ces expériences le démontrent, il est difficile de trouver un consensus entre les experts sur chacune des trois étapes du processus.

La suite de ce chapitre est composée comme suit. La section 4.1 décrit l'expérience construite pour comprendre la stratégie suivie par les experts quant à la sélection des fonctionnalités qui doivent être présentées dans la documentation d'API. La section 4.2 explique l'expérience menée pour comprendre la stratégie pour la création de groupes de fonctionnalités représentant les sections et sous-sections de la documentation d'API. La section 4.3 présente l'expérience analysant la stratégie suivie pour ordonner le contenu des sections et sous-sections. Nous listons les menaces à la validité de notre étude dans la section 4.4. La section 4.5 présente nos conclusions générales sur la difficulté d'une approche complètement automatisée à la lumière des résultats obtenus lors de nos expériences. Elle justifie notre choix de nous orienter vers une approche *semi-automatisée* mettant l'expert au centre du processus de décision.

4.1 Sélection de contenu automatisée

La sélection du contenu consiste à choisir les fonctionnalités du programme ou de la bibliothèque qui doivent figurer dans le fichier README. Dans le cas d'une sélection automatique, ces fonctionnalités sont extraites directement à partir des entités du modèle telles les modules, les classes et les méthodes.

Dans cette section, nous cherchons à comprendre la méthodologie suivie par les écrivains de documentation quant à la sélection des entités du modèle qui doivent figurer dans le fichier README.

Nous avons établi une expérience impliquant des experts du langage Nit ou de ses bibliothèques visant à comparer les entités qu'ils jugent importantes de faire figurer dans la documentation.

4.1.1 Expérience réalisée

Nous avons recruté six contributeurs du langage Nit ayant déjà documenté et publié une bibliothèque ou un programme dans le projet. Chacun de ces experts figure parmi les contributeurs les plus actifs du projet.

Nous avons fourni à chaque expert la liste des entités d'une bibliothèque dont il est l'auteur ou le contributeur et lui avons demandé de supprimer les entités qu'il ne juge pas nécessaire de présenter à l'utilisateur.

Nous avons sélectionné 14 bibliothèques figurant dans le dépôt officiel du langage. Ces bibliothèques ont été choisies de manière à représenter un large éventail de domaines d'application allant d'une bibliothèque de fonctions simple (`md5`) à la bibliothèque standard d'un langage (`core`) ou encore à un cadre de développement pour applications Android (`android`). Le détail des bibliothèques sélectionnées est donné dans le tableau 4.1.

Au total, les 14 bibliothèques sélectionnées représentaient 138 modules, 748 classes et 4107 propriétés. Chaque expert a évalué entre deux et quatre bibliothèques :

expert 1 : 184 entités (`graphs`, `trees`)

expert 2 : 1.061 entités (`android`, `json`)

expert 3 : 1.557 entités (`github`, `markdown`, `mongodb`, `neo4j`)

expert 4 : 1.760 entités (`core`, `model`)

expert 5 : 1.268 entités (`core`, `crypto`, `md5`, `sha1`)

expert 6 : 1480 entités (`android`, `json`, `sqlite`)

Tableau 4.1: Détails des bibliothèques sélectionnées.

Bibliothèque	Entités	Description
`android`	1061	Framework de programmation d'applications Android
`core`	1244	Bibliothèque standard de Nit
`crypto`	11	Bibliothèque de cryptographie
`json`	329	Bibliothèque de manipulation JSON et de sérialisation
`github`	475	Bibliothèque de communication pour l'API REST de GitHub
`graphs`	84	Bibliothèque de modélisation et de manipulation de graphes
`markdown`	542	Compilateur Markdown
`md5`	7	Bibliothèque d'encodage MD5
`model`	516	Modèle objet utilisé par la chaîne de compilation Nit
`mongodb`	190	Connecteur de bases de données MongoDB
`neo4j`	350	Connecteur de bases de données graphes Neo4j
`sha1`	6	Bibliothèque d'encodage SHA1
`sqlite3`	90	Connecteur de base de données graphes SQLite3
`trees`	102	Bibliothèque de modélisation et de manipulation d'arbres

```
package
 module
  Classe
  _attribut
  +methode_publique
  -methode_privee
  ~methode_protegee
```

Figure 4.1: Format des listes d'entités fournies aux experts.

La liste des entités fournie à chaque expert suivait le format décrit dans la figure 4.1. Afin de faciliter la lecture des experts, chaque classe et propriété était préfixée selon son modificateur de visibilité :

+ méthode publique

- méthode privée

~ méthode protégée

_ attribut [1]

De plus, nous avons supprimé a priori les entrées relatives au code généré par les outils de la chaîne de compilation, aux tests unitaires et aux exemples. Nous avons aussi filtré les entités importées ou héritées mais non-redéfinies.

Nous avons ensuite demandé aux experts d'analyser les listes que nous leurs fournissons et de supprimer les entités qu'ils ne souhaitent jamais présenter à l'utilisateur dans la documentation d'API.

4.1.2 Résultats

Au total, les six experts ont analysé 17 listes issues de la bibliothèque standard du langage Nit et de la chaîne de compilation. Ceci représentait un ensemble de 7312 entités à sélectionner.

---

1. Notons qu'en Nit, les attributs sont toujours privés, mais des accesseurs en lecture et écriture sont générés automatiquement.

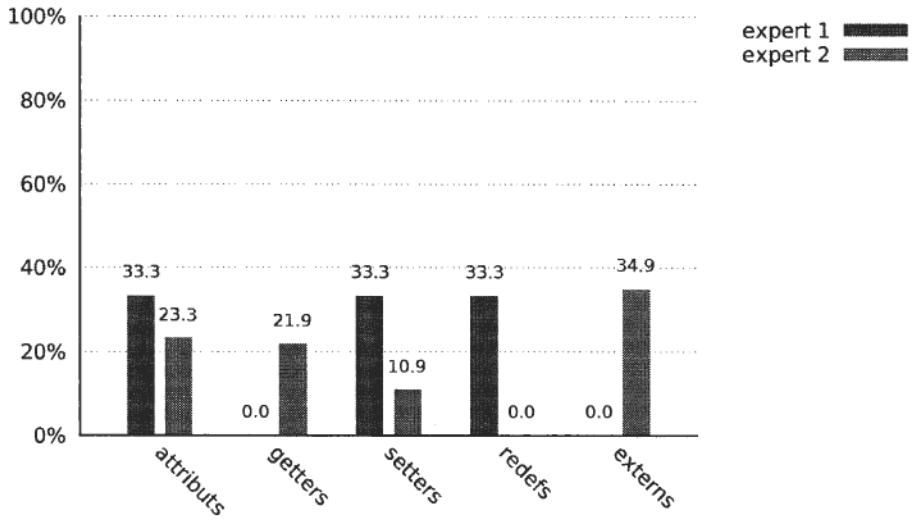


Figure 4.2: Pourcentage des propriétés supprimées par chaque expert en fonction du type.

Sur les 7312 entités, seules 194 (2.7%) ont été supprimées par les experts. Seulement deux experts sur les six ont jugé utile de filtrer des entités. L'expert 1 a supprimé 36 entités (soit 18.5% de toutes les entités supprimées par les experts) et l'expert 2 a supprimé 158 entités (81.5%). À ce stade, nous pouvons confirmer que supprimer définitivement des entités de la documentation ne trouve pas de consensus parmi les experts.

Afin de comprendre la stratégie suivie par les experts lors de la suppression des entités, nous avons analysé les types d'entités supprimées à savoir si elles sont des modules, classes ou des méthodes. 100% des entités supprimées sont des propriétés (attributs ou méthodes). Le détail des propriétés supprimées par chaque expert est donné en figure 4.2.

Nous avons constaté que l'expert 1 a systématiquement retiré les attributs (12 attributs), les accesseurs automatiques en écriture (12 *setters*) pour ne présenter que les accesseurs en lecture (*getters*). Les redéfinitions, équivalentes du `@Override` en

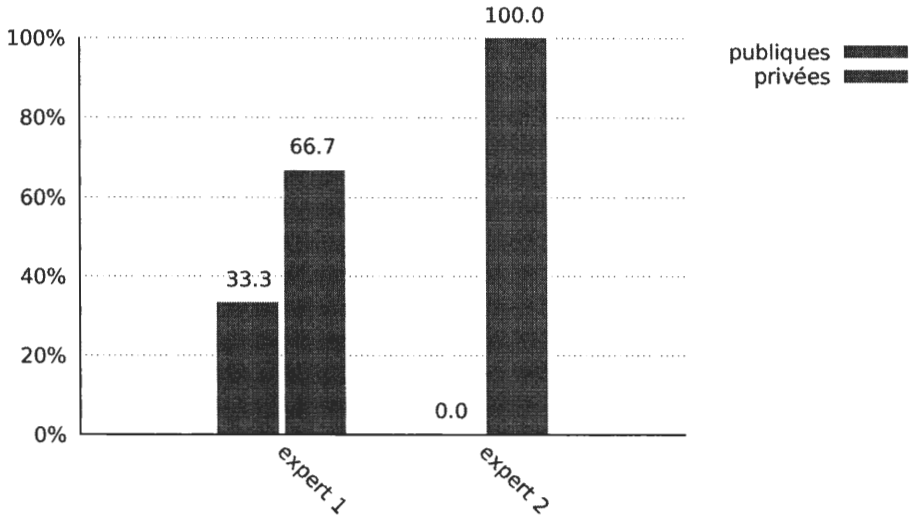


Figure 4.3: Pourcentage des propriétés supprimées par chaque expert en fonction de la visibilité.

Java, sont aussi supprimées (12 *redefs*). Autre fait notable : 100% des redéfinitions supprimées l'étaient par l'expert 1. Les autres experts ont tous jugé préférable de faire figurer les redéfinitions dans la documentation de l'API.

Pour l'expert 2, certains attributs ont été conservés mais ce sont les accesseurs automatiques en lecture (32 *getters*) et en écriture (29 *setters*) qui ont été systématiquement supprimés. L'expert 2 avait aussi supprimé toutes les méthodes externes qui lui étaient présentées (51 *externs*). Ces méthodes sont implémentées en langage C au travers de l'interface de fonctions étrangères du langage Nit (Laferrière, 2018). Elles représentent généralement des fonctionnalités bas niveau telles que l'accès au système.

Enfin, la figure 4.3 donne la répartition des propriétés supprimées en fonction de leur visibilité pour chaque expert. Au total, 93% des propriétés supprimées étaient privées (170 propriétés). Les 12 entités publiques supprimées l'étaient par l'expert 1 et représentaient les 12 redéfinitions supprimées.

Nous avons interrogé les experts 1 et 2 après l'analyse des résultats afin de mieux comprendre leur raisonnement dans la suppression des entités.

Expert 1 À la question « Pourquoi supprimer les attributs et accesseurs en écriture mais laisser les accesseurs en lecture ? », sa réponse était que les attributs et les accesseurs en lecture représentent un détail d'implémentation relatif au langage Nit. Puisque les attributs sont toujours inaccessibles et que les accesseurs en écritures sont privés par défaut, le choix de l'expert était d'alléger la documentation en ne présentant que la partie utilisable : l'accesseur en lecture. Le reste de l'information se trouve directement dans le code et ne nécessite pas de documentation supplémentaire.

À la question « Pourquoi supprimer les redéfinitions de méthodes ? », sa réponse était que cette information n'a que peu d'intérêt pour le lecteur qui souhaite utiliser la bibliothèque. Pour un mainteneur, l'information ne parait pas pertinente non plus car il s'agit d'un détail d'implémentation facilement identifiable dans le code et que l'expert n'a jamais commenté les redéfinitions.

Expert 2 À la question « Pourquoi avoir supprimé certains attributs et accesseurs mais pas tous ? », sa réponse était qu'il a supprimé systématiquement toutes les méthodes externes en rapport avec des fonctionnalités bas niveau implémentées en langage C. Ceci correspondait aux méthodes externes et aux attributs et accesseurs en rapport avec le code externe.

4.1.3 Discussion sur les limites d'une sélection automatisée

Au final, seuls 2% des entités ont été filtrées par les experts et seuls deux experts sur les six ont jugé nécessaire de filtrer des entités. En revanche, les deux experts

ayant supprimé des entités l'ont fait en suivant une approche systématique : retirer les attributs, les redéfinitions et le code externe.

Le besoin de filtrer des entités ne fait donc pas unanimité chez les experts consultés et tous ne sont pas d'accord sur quelles entités devraient être filtrées. Il est donc difficile à ce stade d'établir une approche automatique satisfaisant tous et chacun.

4.2 Regroupement de contenu automatisé

Afin de faciliter la lecture de la documentation, la présentation des fonctionnalités dans le fichier README doit être structurée au moyen de sections et sous-sections. Quelle que soit la stratégie suivie pour sélectionner les entités du modèle à présenter, nous nous demandons maintenant comment grouper ces entités.

Dans cette section, nous cherchons donc à établir une stratégie systématique permettant d'organiser les fonctionnalités à présenter sous forme de groupes qui représenterons les sections et sous-sections dans le README. Pour ce faire, nous avons effectué une expérience impliquant les mêmes six experts que dans notre expérience précédente (section 4.1).

4.2.1 Expérience réalisée

Nous avons fourni la liste des entités présentes dans chaque bibliothèque à l'expert l'ayant écrit en réutilisant le format de l'expérience sur la sélection des entités (section 4.1). L'ordre des lignes dans le fichier était mélangé de manière aléatoire tout en conservant la hiérarchisation par modules et classes.

Cette fois, nous avons demandé à chaque expert de réarranger les entités présentes dans le fichier en groupes de fonctionnalités connexes représentant les sections et sous-sections dans la documentation finale.

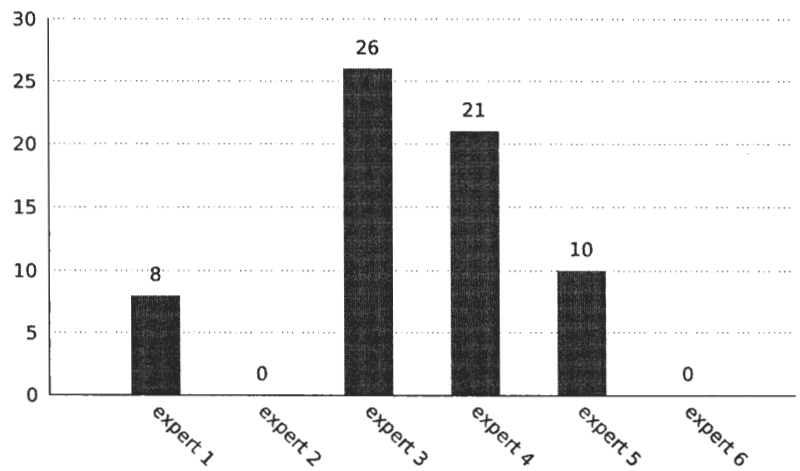


Figure 4.4: Nombre de groupes créés par chaque expert.

Les groupes ont été ajoutés grâce à la syntaxe Markdown pour les sections, c'est-à-dire en préfixant le titre de la section par une séquence de un à six croisillons (#) représentant le niveau de la section ou sous-section.

Nous avons ensuite analysé la liste telle que modifiée par l'expert pour comprendre la stratégie utilisée pour créer les groupes d'entités.

4.2.2 Résultats

Après le travail effectué par les experts, sur les 17 listes fournies, seules 6 contenaient des groupes pour un total de 65 groupes toutes listes confondues. La figure 4.4 compte le nombre de groupes créés par chaque expert. Sur les six experts, deux ont choisi de ne pas créer de groupes.

Dans un questionnaire adressé aux experts après l'expérience, nous avons demandé pourquoi certains experts n'avaient pas créé de groupes et pourquoi certaines listes ne contenaient aucun groupe là où le même expert a choisi d'ajouter des groupes dans une autre liste.

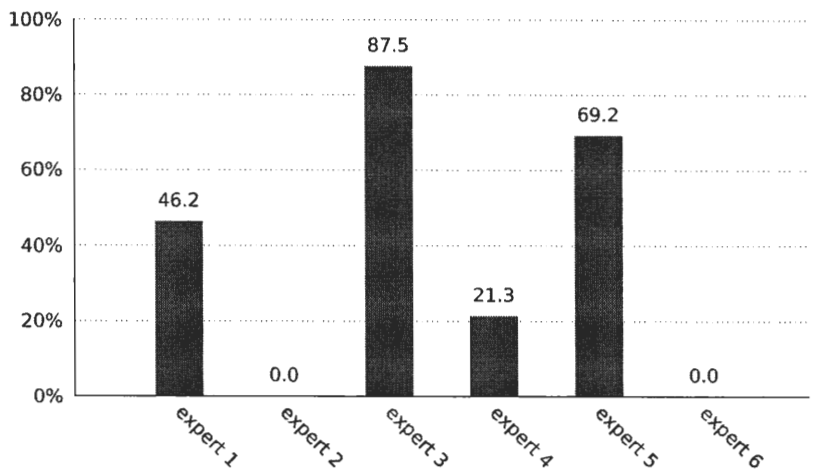


Figure 4.5: Pourcentage d'entités placées dans des groupes pour chaque expert.

Pour les experts 2 et 6, le code Nit est déjà regroupé selon la hiérarchie des modules et des classes et ne nécessite donc pas d'autre type de regroupement. En effet, selon Ducournau *et al.* (2008), la programmation par modules et raffinement de classes telle que proposée en Nit permet de regrouper les fonctionnalités par préoccupations communes de façon plus efficace que l'organisation par packages à la Java.

Les résultats de l'étude de Robillard et DeLine (2010) sur les obstacles à la compréhension des API montrent que la structure d'une documentation peut être déconcertante pour le lecteur quand elle n'est pas alignée avec celle du code. On peut voir en cette observation une explication au fait que certains de nos experts préfèrent calquer leur discours à la structure par modules.

Pour les experts 1, 3, 4 et 5, certaines listes étaient trop courtes (moins d'une vingtaine de fonctionnalités) pour nécessiter un regroupement au delà de celui des classes. Ainsi, seules 6 listes sur les 17 contenaient un regroupement créé par les experts. De plus, l'approche n'était pas non plus systématique au sein d'une même liste. La figure 4.5 montre le pourcentage d'entités placées dans des groupes

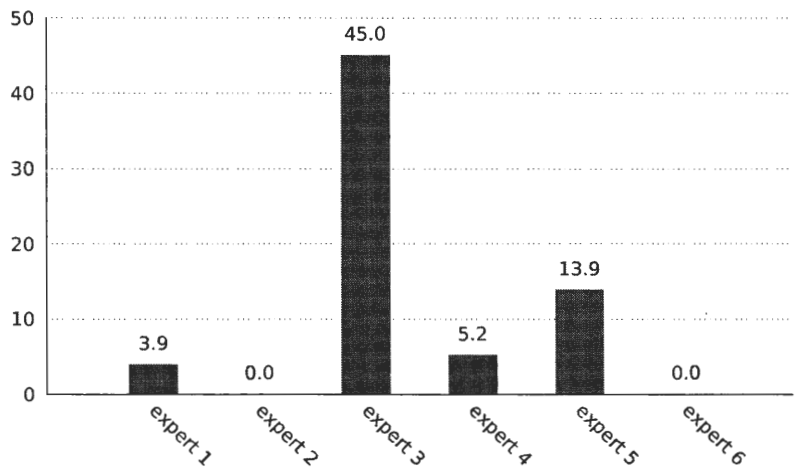


Figure 4.6: Nombre moyen d'entités par groupe pour chaque expert.

seulement pour les listes contenant effectivement des groupes. On remarque ici que toutes les entités n'étaient pas forcément regroupées. Sur l'ensemble des 6 listes contenant des groupes, seuls 21% à 87% des propriétés étaient placées dans des groupes. Les autres étaient laissées selon l'organisation du code.

Déjà à ce stade, l'idée de regrouper les fonctionnalités issues du code et l'approche à suivre pour les regrouper ne fait pas consensus.

Le nombre d'entités par groupe ne fait pas non plus consensus. Comme le montre la figure 4.6, certains experts (1, 4 et 5) ont préféré des groupes contenant peu de fonctionnalités (moyennes de 3.9 à 13.9 fonctionnalités par groupe) alors que d'autres (l'expert 3) ont composé des groupes plus gros (45.0 entités en moyenne par groupe).

La composition des groupes par type d'entité n'était pas non plus régulière entre les experts. Les experts 1 et 4 ont choisi de ne regrouper que des propriétés au sein des classes. Quand ils ont été interrogés sur leur méthodologie, ces deux experts ont estimé que le regroupement par modules et classes était suffisant pour former des

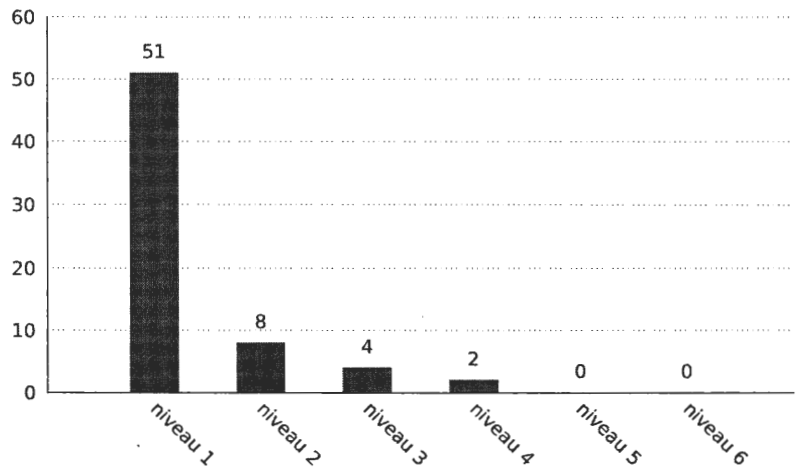


Figure 4.7: Nombre de groupes créés pour chaque niveau de section.

groupes d'un haut niveau d'abstraction mais que certaines propriétés méritaient d'être regroupées selon un niveau de détails.

À l'opposé, l'expert 5 a formé ses groupes au niveau des packages, des modules et des classes et a choisi d'ignorer complètement les propriétés. Selon son questionnaire, il a préféré regrouper les fonctionnalités selon un haut niveau d'abstraction et laisser le détails des propriétés selon une organisation par classes.

L'expert 3 a mélangé à la fois les modules, les classes et les propriétés dans ses groupes. Selon son questionnaire, le type de regroupement dépendait avant tout de ce que l'on veut expliquer ou mettre en valeur dans la documentation.

En général, les experts ont préféré conserver l'organisation du code en suivant la hiérarchie des packages, modules et classes et ont créé leurs groupes au sein de cette hiérarchie. Seuls deux experts (3 et 5) ont déplacé des entités en dehors de leur hiérarchie d'origine, totalisant respectivement 32 (3.5%) et 25 (1.2%) des fonctionnalités trouvées dans les groupes.

L'imbrication des groupes était généralement limitée. Selon la figure 4.7, 51 groupes sur les 65 créés étaient de niveau 1 (78.5%), 8 groupes étaient de niveau 2 (12.3%),

4 groupes de niveau 3 (6.1%), 2 groupes de niveau 4 (3.1%). Il n'existait aucun groupe de niveau 5 et 6. Seuls les experts 3 et 5 ont décidé d'utiliser des groupes imbriqués, tous les autres se sont contenté d'un seul niveau.

Notons ici que les niveaux des groupes formés par les experts sont sans rapport avec les niveaux des sections traditionnelles des fichiers README que nous avons pu observer au chapitre 2. En effet, selon les paramètres de notre expérience, il était demandé aux experts de former des groupes à partir du contenu qui leur était fourni sans prendre en compte d'autres sections éventuellement présentes dans la documentation en cours de rédaction.

4.2.3 Discussion sur les limites d'un regroupement automatisé

Créer des groupes d'entités à partir du contenu du modèle est une tâche complexe même pour l'expert à l'origine du code. Notre expérience a montré qu'il est difficile d'obtenir un consensus aussi bien sur le type d'entités à regrouper que sur le contenu des groupes.

À ce stade, nous ne pouvons isoler la stratégie suivie par les experts pour confectionner les sections et sous-sections de leur documentation. Les experts ne suivent pas tous la même approche dans le regroupement des fonctionnalités et ne sont pas d'accord quant à l'objectif de ces regroupements.

4.3 Ordre du contenu automatisé

Contrairement à une documentation d'API, un README présente un contenu structuré et ordonné. Les fonctionnalités sont présentées une à une selon une suite logique dans un ordre qui favorise la compréhension par le lecteur. Quelle que soit la stratégie suivie pour sélectionner puis regrouper les entités du modèle à présenter, nous nous demandons maintenant dans quel ordre présenter les entités.

Dans cette section, nous cherchons donc à établir une stratégie systématique permettant d'ordonner les fonctionnalités à présenter dans le README. Pour ce faire, nous avons effectué une expérience impliquant à nouveau nos six experts tel que dans les expériences sur la sélection des entités (section 4.1) et sur leur regroupement (section 4.2).

4.3.1 Expérience réalisée

Nous avons fourni la liste des entités présentes dans chaque bibliothèque à l'expert l'ayant écrit en réutilisant le format de l'expérience sur la sélection des entités (section 4.1). L'ordre des lignes dans le fichier était mélangé de manière aléatoire tout en conservant la hiérarchisation par modules et classes. Nous avons ensuite demandé à l'expert de réordonner la liste selon l'ordre qui lui semblait le plus logique pour expliquer sa bibliothèque.

Afin de comprendre la stratégie suivie par les experts-écrivains pour choisir l'ordre des entités à documenter, notre idée consistait à comparer chaque ordre écrit à une série d'ordres générés par des algorithmes.

Nous avons sélectionné 10 algorithmes permettant de comparer et ordonner des entités d'un modèle que nous avons trouvé soit dans les outils d'auto-documentation actuels, soit dans la littérature. Nous regroupons ces ordres en quatre catégories d'approches que nous décrivons ci-dessous.

4.3.1.1 Approches naïves

Nous regroupons sous le nom *Approches naïves* les algorithmes d'ordre qui utilisent une logique simple.

Ordre lexicographique (`lex`) Les entités à documenter sont triées selon leur nom suivant un ordre lexicographique. C'est l'approche utilisée par la plupart des outils de documentation, par exemple, `Javadoc` ou `Rubydoc` pour n'en citer que deux.

Ordre naturel (`naturel`) Les entités à documenter sont triées selon leur ordre d'apparition dans le code source pour chaque unité de compilation. Les unités de compilation sont triées selon l'ordre lexicographique des noms de fichiers (comme ils le sont dans un explorateur de fichier ou un IDE). Par exemple, un tel ordre de tri peut être activé par l'intermédiaire d'une option dans les outils `Doxygen` et `Rubydoc`.

Un cas particulier doit être expliqué pour le langage Nit. En effet, afin d'alléger le travail du développeur, certaines propriétés sont générées automatiquement par le compilateur Nit. Par exemple, les constructeurs par défaut sont générés à partir des attributs présents dans les classes. Il en est de même pour les *getters* et *setters* des attributs. Dans le cas où une propriété à trier ne possède pas de référence directe vers le code, nous modifions cet algorithme de tri pour prendre en compte la position de l'entité à l'origine de la génération (comme la position de l'attribut par exemple).

4.3.1.2 Approches par dépendances

Les approches par dépendances utilisent les relations d'importation ou d'utilisation pour ordonner les entités du modèle.

Ordre structurel (`struct`) Les entités à documenter sont triées selon leur apparition dans l'ordre d'importation des modules et d'héritage des classes : les modules importés sont présentés avant les modules qui les importent, alors que les super-classes sont présentées avant leurs sous-classes. Cet algorithme est celui présenté par `Scaladoc` aux lecteurs. Nous utilisons l'ordre de linéarisation du compilateur pour résoudre l'ordre d'affichage des classes en héritage multiple. Les propriétés introduites sont présentées avant leurs redéfinitions.

Ordre d'usage (`usage`) Les entités à documenter sont triées selon leur utilisation dans le code source. Soit deux entités du modèle, A et B. Si A utilise B alors B sera présentée avant A. La notion d'utilisation varie en fonction du type d'entité considéré. Ainsi, nous définissons une utilisation de module comme le fait d'utiliser une classe ou une propriété définie dans ce module. Une utilisation de classe consiste à instancier cette classe, appeler une de ses propriétés ou utiliser le type de la classe dans une signature. Utiliser une propriété consiste à l'invoquer.

Nous sélectionnons cet algorithme car il permet de représenter l'ordre d'utilisation des entités lors de l'exécution comme un développeur le suivrait dans un débogueur.

Ordre d'usage dans les commentaires (`doc`) Les entités à documenter sont triées selon leur utilisation dans les commentaires des autres entités. Soit deux entités du modèle, A et B. Si le commentaire de A fait référence à l'entité B alors B sera présentée avant A dans la documentation. Faire référence à une entité dans un commentaire consiste à écrire son nom court ou son nom canonique — comme `Array` ou `core::Array` — dans le texte du commentaire.

Nous utilisons cet algorithme pour présenter en premier les entités qui ne nécessitent pas la compréhension d'autres entités et limiter au maximum les références vers l'avant.

4.3.1.3 Approches par quantification

Les approches par quantification utilisent des métriques représentant la taille des entités à documenter pour les ordonner.

Ordre par nombre de lignes de code (`loc`) Les entités à documenter sont triées selon leur nombre de lignes de code sans les commentaires, et les plus grosses entités sont présentées en premier. Avec cet algorithme, nous suivons l'idée que les plus grosses entités dans le code source sont en fait les plus importantes à expliquer.

Ordre par nombre de définitions (`defs`) Les entités à documenter sont triées selon leur nombre de définitions. Les modules sont triés selon le nombre de définitions de classes qu'ils contiennent. Les classes sont triées en fonction du nombre de propriétés. Les propriétés sont triées en fonction du nombre de redéfinitions. Là encore nous suivons l'idée que les plus grosses entités sont à documenter en premier, mais faisons abstraction du code source lié à ces entités.

Ordre par nombre de lignes de documentation (`lod`) Les entités à documenter sont triées selon le nombre de lignes du cartouche de commentaire. Les entités les plus documentées sont donc présentées en premier. Avec cette approche, nous partons du principe que les plus longs commentaires contiennent une grande quantité d'information utiles pour la compréhension des autres entités.

4.3.1.4 Approches par importance

Nous sélectionnons enfin deux approches permettant d'allouer des scores d'importance aux entités du modèle.

MENDEL (`mendel`) L'approche MENDEL proposée par Denier et Guéhéneuc (2008) permet de sélectionner les classes les plus importantes d'un programme ou d'une bibliothèque en fonction de leur contribution en terme d'introduction de propriétés.

L'approche est basée sur la métrique *Class Novelty Index* (*nvi*) définie comme suit :

$$bms(c) = \frac{|TotS(c)|}{DIT(c)+1} \qquad nvi(c) = \frac{LocS(c)|}{bms(parents(c))}$$

Ici, $bms(c)$ représente la taille moyenne de la branche (*Branch Mean Size*) calculée en divisant le nombre de propriétés héritées et introduites dans la classe c par la profondeur dans l'arbre d'héritage (*Depth of Inheritance Tree*) de c. La valeur de $nvi(c)$ est obtenue en divisant le nombre de propriétés introduites dans la classe c par le *bms* moyen des parents directs de c.

Le *Class Novelty Index* obtenu ainsi est donc indépendant de la taille de la hiérarchie de classes. Nous utilisons cet index comme un score d'importance pour présenter les entités avec le *nvi* le plus grand en premier. La même formule peut être appliquée aux modules en prenant en compte le nombre d'introductions de classes. En revanche, cette approche ne s'applique pas au tri des propriétés.

PageRank (`prank`) Conçu par Page *et al.* (1999) et traditionnellement utilisé par le moteur de recherche Google, l'algorithme PageRank permet d'ordonner les nœuds d'un graphe par leur importance qui dépend du nombre d'arêtes entrantes à chaque nœud.

La valeur PageRank d'un nœud u est obtenue par la formule suivante, avec $0 < c < 1$:

$$PR(u) = c \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

$PR(u)$ représente la somme des valeurs PageRank de chaque nœud v parmi le set B_u des noeuds pointant vers u divisée par le nombre d'arêtes $L(v)$ sortantes de v. c est une constante définie manuellement, sa valeur est généralement de 0.1. La définition PR est récursive, mais une solution peut être obtenue par un processus itératif jusqu'à l'obtention d'un point fixe.

Cet algorithme est utilisé par l'approche CodeRank de Neate *et al.* (2006) et permet d'ordonner les classes d'un programme ou d'une bibliothèque en leur attribuant une valeur PageRank. Dans CodeRank, les classes sont considérées comme les nœuds d'un graphe, et les arêtes représentent les relations de spécialisation et d'utilisation. Nous appliquons cette même approche aux modules, en prenant en compte les relations d'importation, et aux propriétés, en fonction de l'ordre d'appel.

4.3.1.5 Comparaison des ordres

Notre idée était de générer des ordres à l'aide des algorithmes décrits précédemment puis de les comparer aux ordres écrits par les experts. Il nous fallait donc pouvoir mesurer la similitude — ou le degré de différence — entre deux ordres. Pour ce faire, nous avons sélectionné deux métriques permettant de comparer la distance entre deux séquences.

Distance de Levenshtein Cette métrique est traditionnellement utilisée pour mesurer la distance d'édition entre deux chaînes de caractères (Gusfield, 1997). La métrique associe un coût à chaque opération d'insertion, de suppression ou de substitution de caractère nécessaire pour transformer une chaîne en une autre. Nous choisissons ici un coût de 1 pour chaque opération. L'objectif est alors de

trouver le nombre minimum d'opérations permettant de transformer une chaîne en une autre.

Soit la chaîne ABC. La distance de Levenshtein entre ABC et elle-même est de 0, car aucune opération n'est nécessaire.

Soit les chaînes ABC et CBA. La distance de Levenshtein entre ces deux chaînes est de 2 : il faut au minimum deux opérations pour passer de CBA à ABC : 1) substituer le C par un A ; 2) substituer le A par un C.

La distance d'édition peut être transformée en un score de similarité comme suit, où 100% représente une similarité parfaite entre les deux listes et 0% indique deux listes aux ordres totalement différents :

$$(1 - distance/taille(liste)) \times 100$$

Avec $taille(liste)$ permettant de normaliser la distance par rapport au nombre maximal d'opérations possibles sur la liste.

Distance de Kendall-Tau La distance de Kendall-Tau (aussi appelée distance de rang ou distance de permutation, ou encore *Bubble-Sort distance* en anglais) mesure la distance entre deux listes en comptant le nombre de permutations nécessaires pour passer d'une liste à l'autre (Kendall, 1938). La métrique associe un coût de 1 à chaque permutation.

Soit la chaîne ABC. La distance de Kendall-Tau entre ABC et elle-même est de 0 : aucune permutation n'est nécessaire.

Soit les chaînes ABC et CBA. La distance de Kendall-Tau entre ces deux chaînes est de 3 : il faut trois permutations pour passer de CBA à ABC : 1) permuter C et B pour obtenir BCA ; 2) permuter C et A pour obtenir BAC ; 3) permuter B et A pour obtenir ABC.

Selon Fagin *et al.* (2003), la distance de permutation peut être transformée en un score de similarité comme suit, où 100% représente une similarité parfaite et 0% indique deux listes aux ordres totalement différents :

$$(1 - distance/(taille(liste) \times (taille(liste) - 1))) \times 100$$

Avec $taille(liste) \times (taille(liste) - 1)$ représentant le nombre maximal de permutations possibles pour la liste.

4.3.2 Résultats

Nous avons comparé les ordres obtenus par les algorithmes présentés avec les ordres produits par les experts. Pour chaque ordre produit par un expert, nous avons repris la liste des entités qu'il avait à trier et l'avons réordonné de façon aléatoire. Nous avons ensuite appliqué chacun des 10 algorithmes de tri sur la liste et avons ainsi obtenu 10 ordres à comparer avec chaque liste triée par un expert.

Pour chaque ordre obtenu, nous avons mesuré sa distance par rapport à l'ordre produit par l'expert en utilisant les métriques Levenshtein et Kendall-Tau. Nous avons ensuite compilé la moyenne des distances pour chaque algorithme afin d'obtenir la similarité moyenne des ordres générés par chaque approche par rapport aux ordres souhaités par les experts.

4.3.2.1 Ordre aléatoire

Nous avons commencé par analyser la performance d'un algorithme de tri aléatoire en le comparant aux ordres produits par les experts. Bien que peu utile, l'ordre aléatoire nous permettait tout d'abord de vérifier nos techniques de comparaison d'ordres en évaluant la probabilité de trouver le même ordre qu'un expert aléatoirement.

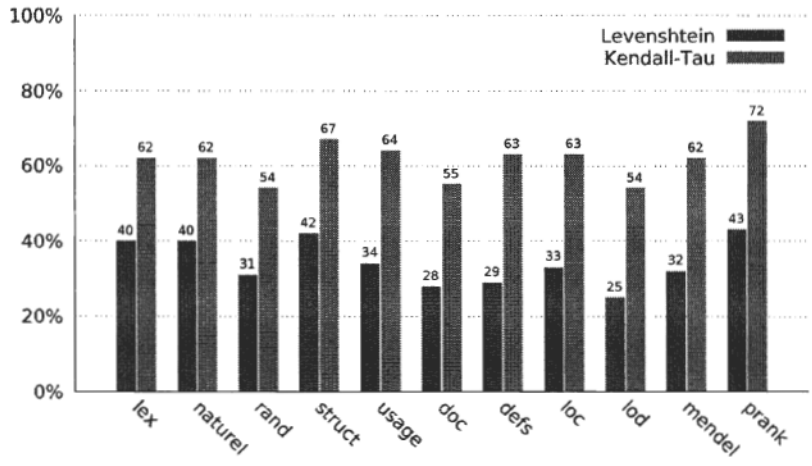


Figure 4.8: Similarités moyennes obtenues sur les ordres des modules avec les distances de Levenshtein et Kendall-Tau.

La distance de Levenshtein donnait une similarité moyenne de 23% avec les ordres produits par les experts. Ceci signifie que seulement 23% des entités dans les ordres sont correctement placés par rapport aux choix des experts, les 77% restants étant considérés comme étant à la mauvaise place.

La distance de Kendall-Tau montrait une similarité moyenne de 47%. Ceci indique que 47% des paires d'entités sont correctement placées l'une par rapport à l'autre mais que l'ordre général n'est pas respecté.

4.3.2.2 Ordre des modules

Nous avons ensuite comparé les résultats produits par les approches sélectionnées sur l'ordre des modules par rapport aux 17 ordres concernant les modules produits par les experts. Notons que les tris par ordre lexicographique et par ordre naturel étaient identiques quand ils étaient appliqués aux modules (ordre lexicographique des noms de fichiers). La figure 4.8 donne le pourcentage de similarité moyen de chaque approche selon les distances de Levenshtein et de Kendall-Tau.

La similarité moyenne selon Levenshtein était de 34%. Seulement trois approches offraient des résultats plus haut que la moyenne : les tris par ordre lexicographique (`lex`), par ordre de dépendances structurelles (`struct`) et par PageRank (`prank`). Les ordres basés sur la taille (nombre de lignes de code ou de documentation, nombre de définitions) étaient les moins performantes, même moins performantes que l'approche aléatoire en moyenne.

Les résultats obtenus avec la distance de Kendall-Tau étaient plus mitigés. La moyenne des distances est de 62%. Seules les approches par ordre de dépendances structurelles (`struct`) et par PageRank (`prank`) ressortaient clairement.

L'approche PageRank utilise trois critères pour établir son ordre : les relations d'importation de modules, l'utilisation d'un type statique issu d'un module et l'appel d'une propriété introduite par un module (l'instanciation d'une classe étant considérée comme un appel au constructeur). Afin de mieux comprendre l'impact de chacun de ces critères sur la similarité de l'ordre généré, nous avons comparé cinq versions de l'algorithme :

`imports` PageRank uniquement sur les arêtes d'importation de modules.

`types` PageRank uniquement sur les arêtes d'utilisation de types statiques.

`props` PageRank uniquement sur les arêtes d'appels de propriétés.

`types + props` PageRank sur les arêtes d'utilisation de types et d'appels de propriétés, donc correspond à toutes les arêtes d'utilisation d'un module.

`tous` PageRank comprenant toutes les arêtes.

Comme le montre la figure 4.9, c'était la version de PageRank basée sur les importations qui donnait les meilleurs résultats. Ces résultats étaient d'ailleurs comparables à ceux de l'ordre par dépendances structurelles (`struct`), c'est-à-dire selon l'ordre de linéarisation de l'importation des modules.

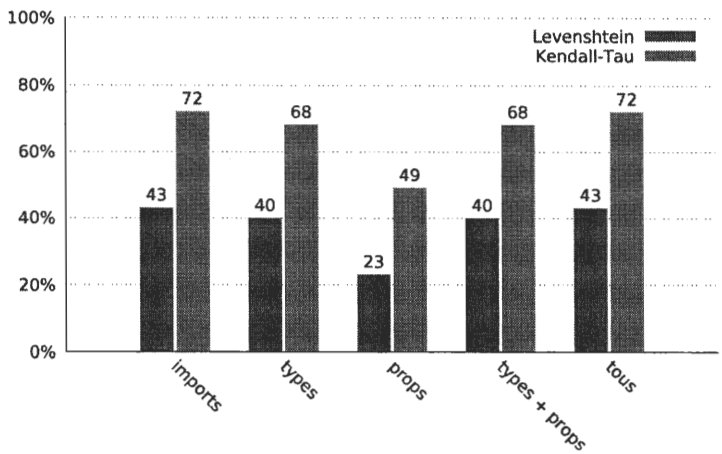


Figure 4.9: Similarités moyennes obtenues par les variations de PageRank sur les ordres des modules avec les distances de Levenshtein et Kendall-Tau.

Finalement, ce sont les approches basées sur l'ordre d'importation des modules qui ont produit les ordres les plus similaires à ceux des experts avec un avantage à PageRank plutôt que l'ordre de linéarisation brut. Les experts préfèrent donc présenter les modules les plus utilisés en premier.

4.3.2.3 Ordre des classes

Nous avons appliqué les approches à l'ordre des classes et les avons comparé aux 85 ordres produits par les experts. Notons que pour les classes, les ordres lexicographique et naturel étaient différents. L'ordre naturel prenait en compte la position de la classe dans le fichier. Comme toutes les classes à trier faisaient partie du même module, il n'y avait pas de comparaison entre plusieurs fichiers à faire. La figure 4.10 montre que c'est l'ordre naturel qui a produit les résultats les plus similaires par rapport à ce qu'attendaient les experts.

Les experts documentent donc leurs classes selon l'ordre de définition dans le code source plutôt qu'en suivant l'ordre d'héritage. En effet, c'est une pratique courante en Nit que d'écrire les classes les plus importantes d'un module au début

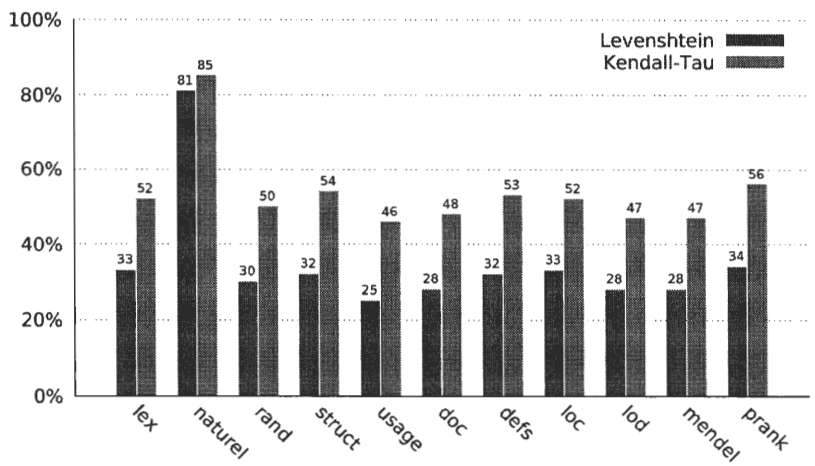


Figure 4.10: Similarité moyenne des ordres des classes selon les distances de Levenshtein et Kendall-Tau.

du fichier. Cet ordre est donc réutilisé par les experts lors de la rédaction de la documentation.

Cette pratique trouve ses origines dans la programmation lettrée (*literate programming*) proposée par Knuth (1992). En effet, avec la programmation lettrée, les programmes sont écrits selon l'ordre logique de compréhension en mélangeant texte libre (commentaires) et code. Le développeur étant donc à la fois écrivain du code et de la documentation, il décide de l'ordre à suivre pour présenter les concepts :

> « [Le programmeur] cherche à obtenir un programme qui est compréhensible parce que les concepts ont été présentés dans le meilleur ordre pour la compréhension humaine, en utilisant un mélange de méthodes formelles et informelles qui se complètent l'une l'autre. »
>
> Donald Knuth, *Literate Programming* (Knuth, 1992)

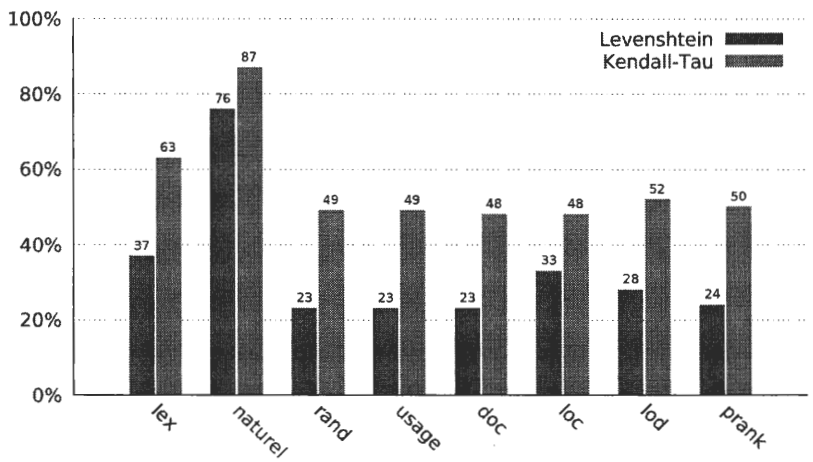


Figure 4.11: Similarité moyenne des ordres des propriétés selon les distances de Levenshtein et Kendall-Tau.

4.3.2.4 Ordre des propriétés

Nous avons ensuite appliqué les approches de tri sur les propriétés et les avons comparé aux 517 ordres produits par les experts-écrivains. Ici, l'ordre sur les dépendances structurelles n'était pas applicable (pas d'importation ou d'héritage) et l'ordre de PageRank était basé seulement sur les appels de propriétés.

Selon les résultats présentés dans la figure 4.11, c'est encore une fois l'ordre naturel qui a produit les résultats les plus proches de ce qu'attendaient les experts. L'ordre alphabétique était aussi favorisé pour trier les entités qui n'étaient pas comparables via leurs dépendances structurelles.

Là encore, c'est l'ordre de déclaration dans le code qui l'a emporté. Le développeur fait donc initialement l'effort d'organiser ses propriétés selon un ordre logique afin de faciliter le travail du lecteur.

4.3.2.5 Ordre global

Nous avons cherché à appliquer un ordre sur la hiérarchie de toutes les entités à documenter pour un package. Selon les observations des sections précédentes, c'était l'ordre structurel basé sur l'importation (`struct` et `prank-imports`) qui donnait le meilleur ordre pour les modules. L'ordre naturel (`nat`) performait le mieux pour les classes et les propriétés.

La hiérarchie des entités d'un package représente un arbre de modules contenant des classes, elles-mêmes contenant des propriétés. Nous avons défini un nouvel algorithme pour ordonner cet arbre en combinant l'ordre structurel d'importation et l'ordre naturel (`struct + nat`). Nous avons utilisé cet algorithme pour ordonner l'arbre des entités d'un package et l'avons comparé aux autres approches par rapport à l'ordre attendu par les experts. Afin de comparer la similarité de deux arbres, nous avons utilisé la distance d'édition entre deux arbres (*Tree Edit Distance*) (Zhang et Shasha, 1989).

Tree Edit Distance La distance d'édition entre deux arbres correspond au nombre minimal de nœuds à modifier pour passer d'un arbre à l'autre. Elle se compare avec la distance de Levenshtein mais pour les arbres plutôt que pour les chaînes. L'approche consiste à pondérer et additionner les modifications faites sur les nœuds en comptant les insertions et suppressions de nœuds ainsi que les modifications d'un nœud en un autre. La distance retournée est un entier positif représentant le nombre de modifications à opérer : une distance nulle indique deux arbres identiques, et plus la distance est grande, plus les arbres sont différents.

La figure 4.12 donne un exemple de transformation pour passer de l'arbre `A` à l'arbre `D`. Tout d'abord, on supprime le nœud `b` et raccroche ses enfants à son parent (coût de 1). On renomme ensuite le nœud `c` en `b` (coût de 1). Enfin, on

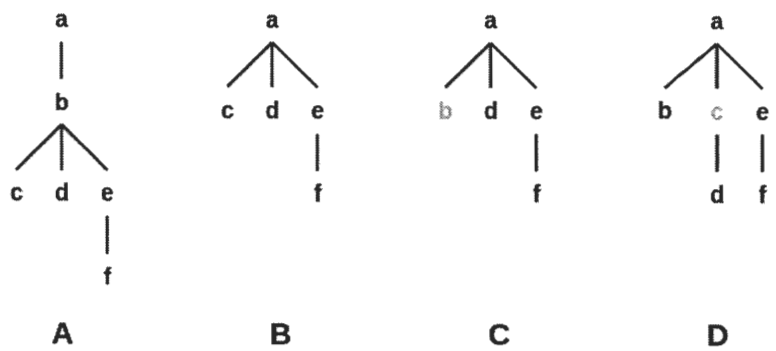


Figure 4.12: Exemple de comparaison d'arbre avec *Tree Edit Distance.*

insère un nouveau nœud `c` comme parent de `d` (coût de 1). La distance entre les deux arbres représente la somme des coûts des modifications. Ici la distance est donc de 3.

Nous traduisons la distance d'édition entre deux arbres en un score de similarité comme pour la distance de Levenshtein où 100% représente une similarité parfaite où tous les éléments sont à la même place dans les deux arbres et 0% indique deux arbres totalement différents :

$$(1 - distance/taille(arbre)) \times 100$$

La figure 4.13 présente la similarité globale des ordres générés par les approches par rapport aux ordres des experts. Comme nous l'attendions, c'est l'ordre de dépendance des modules combiné à l'ordre naturel des classes et des propriétés (`struct + nat`) qui a produit les meilleurs résultats avec 77% de similarité globale.

4.3.3 Discussion sur l'ordre du contenu automatisé

Grâce à l'expérience réalisée dans cette section, nous avons maintenant une meilleure idée de la stratégie générale suivie par les experts pour ordonner les

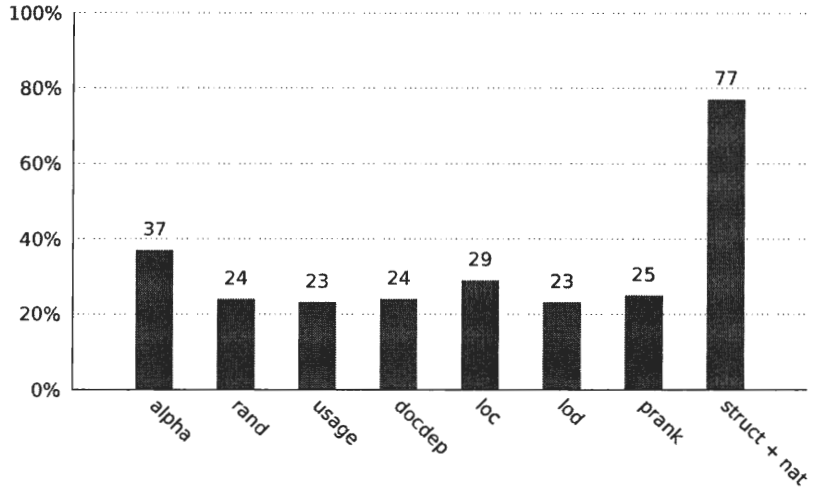


Figure 4.13: Similarité moyenne des ordres complets selon *Tree Edit Distance.*

entités présentées dans le README. En effet, les experts suivent un ordre d'importation pour les modules et l'ordre d'écriture dans l'unité de compilation pour les classes et les propriétés. Une approche automatisée peut donc bénéficier de cette observation en fournissant une stratégie systématique sur l'ordre de présentation des entités.

4.4 Menaces à la validité

Validité de la construction Dans notre étude, nous avons cherché à déterminer le lien entre la structure de la documentation choisie par un expert et le code qu'il documente. Nous avons donc mesuré les structures produites par les experts selon les dimensions de sélection, de regroupement et d'ordre du contenu et avons tenté de les aligner avec les caractéristiques du code.

Une menace possible à la construction de notre étude est le choix des dimensions évaluées. Il est en effet possible que d'autres critères, qui ne sont pas en lien avec le code, soient à prendre en considération dans la compréhension du processus de rédaction de la documentation README relative à l'API.

Validité interne Les experts ayant participé à notre étude sont tous des experts dans l'écriture de bibliothèques Nit ce qui ne veut pas forcément dire qu'il s'agit d'experts dans l'écriture de documentation ou de README.

Il est possible que leurs choix ne soient pas les meilleurs d'un point de vue pédagogique. Cependant, tous les experts sélectionnés sont issus du domaine académique puisque nous comptons deux professeurs, deux chargés de cours et deux auxiliaires d'enseignement, travaillant tous dans le département d'informatique d'une université. Bien qu'une telle répartition ne soit pas représentative de la réalité du monde du génie logiciel, notre échantillon nous semble particulièrement bien construit pour représenter un ensemble d'écrivains pédagogues ayant l'habitude d'expliquer des concepts liés au code source.

Validité externe Il nous parait difficile de généraliser le résultat de nos observations au comportement de tous les écrivains de documentation. En effet, notre échantillon d'étude ne comprend que six experts et ne s'applique qu'à la documentation de projets Nit.

Nit, de par l'utilisation du raffinement de classes permet une organisation du code autour des préoccupations (*concerns*), ce qui n'est pas le cas pour la plupart des autres langages. Les observations faites dans ce chapitre sont donc fortement liées au langage Nit et à ces experts, et ne peuvent pas se généraliser facilement à d'autres experts ou d'autres langages.

4.5 Limites d'une approche automatisée : vers une approche semi-automatisée

Dans ce chapitre, nous avons présenté trois expériences visant à comprendre la stratégie suivie par des experts écrivant une documentation de type README à partir des entités du modèle d'une application ou d'une bibliothèque.

Nous avons découpé le processus d'écriture en trois étapes successives :

1. Sélectionner les fonctionnalités à présenter dans le fichier README ;
2. Regrouper ces fonctionnalités en sections et sous-sections ;
3. Ordonner le contenu des sections et sous-sections.

Notre première expérience portant sur la sélection du contenu (section 4.1) montrait que la stratégie suivie est propre à chaque expert et qu'il est difficile pour eux de choisir a priori les entités qui vont composer le README final. Nous avons néanmoins pu isoler certains comportements comme la suppression des entités privées, des entités externes ou encore des redéfinitions.

La seconde expérience (section 4.2) visait à comprendre la stratégie des experts quant à un éventuel regroupement des fonctionnalités en sections et sous-sections. Nous avons montré que, là encore, le processus suivi diverge selon les experts considérés. Les experts n'étaient pas d'accord sur le type des groupes à créer ou leur contenu. Le choix des sections allant composer le README final est donc difficile à faire a priori.

Dans la troisième expérience (section 4.3), nous avons cherché à mettre en évidence le choix des experts quant à l'ordre de présentation des fonctionnalités dans le README. Selon les résultats de cette expérience, nous avons montré que les experts choisissaient en général de suivre le même ordre de définition que dans le code source à savoir un ordre par importation pour l'ordre des modules et l'ordre d'apparition dans les fichiers pour les classes et les propriétés.

La rédaction d'une documentation README à partir de la liste des entités du modèle est donc une tâche difficile comprenant des décisions qui sont propres à chaque expert. L'absence de consensus entre les experts sur les deux premières étapes que sont la sélection et le regroupement des fonctionnalités montre qu'une approche complètement automatisée permettant de générer le contenu de la documentation

README seulement à partir du modèle de l'application ou de la bibliothèque manquerait à satisfaire chaque expert.

Nous pensons donc qu'une approche *semi-automatisée* d'aide à la rédaction représente une approche préférable à une qui serait complètement automatisée. Les résultats obtenus dans ce chapitre peuvent néanmoins servir de base à une approche semi-automatisée en permettant de mieux sélectionner les options et comportements qui doivent être présentés à l'expert. Ainsi, dans les prochains chapitres, nous décrivons la mise en place d'un système d'aide à la rédaction de fichiers README visant à soutenir les experts dans leur travail de documentation.

Deuxième partie

Documentation d'API pour Nit

CHAPITRE V

`DOC_DOWN` – UNE SPÉCIFICATION ET DES OUTILS DE DOCUMENTATION D'API POUR NIT

Dans ce chapitre, nous présentons `doc_down`, la spécification de documentation d'API pour le langage Nit que nous avons conçue. Cette spécification établit les règles concernant l'architecture des projets et leur documentation. Elle précise le comportement attendu des outils de génération de documentation du langage et les principes à suivre par les développeurs et écrivains de documentation.

Cette spécification est l'une des contributions de notre thèse, et elle est maintenant devenue la spécification officielle de la documentation d'API pour Nit. Sa place dans l'écosystème de document Nit est illustrée à la figure 5.1.

Comme dans la plupart des langages, la documentation d'API des entités du langage Nit est extraite depuis les commentaires et méta-données du code source. Dans le cadre de cette thèse, nous avons raffiné la spécification de la documentation Nit afin de faciliter le travail de l'écrivain et améliorer l'extraction des informations depuis les outils de documentation. Pour chaque aspect de cette spécification, nous avons développé un outil montrant comment utiliser cette information, outil qui sera réutilisé plus tard par les générateurs de documentation du langage Nit (chap. 6).

Cette spécification propose une structure unifiée pour les projets Nit permettant aux développeurs de se retrouver plus rapidement dans un nouveau projet et permettant

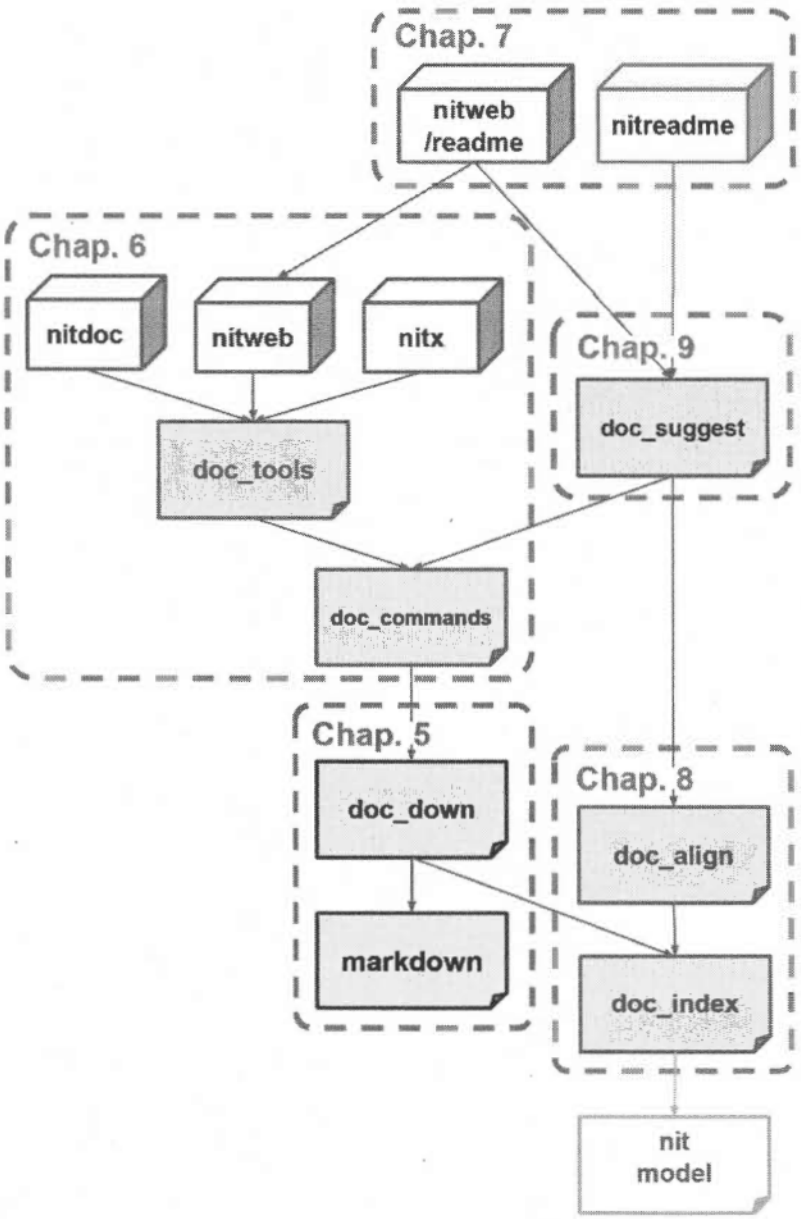


Figure 5.1: Place des modules `markdown` et `doc_down` dans l'écosystème de documentation Nit.

aux outils d'extraire les informations sur le projet simplement (section 5.1). Nous avons développé l'outil `nitpackage` pour automatiser la gestion de cette structure et des méta-données du projet.

Les commentaires des entités sont écrits à l'aide du langage de présentation Markdown, langage simple à utiliser par l'écrivain (section 5.2) qui permet à la fois de structurer la documentation et présenter des exemples de code vérifiables automatiquement (section 5.3). Nous proposons `nitmd`, un compilateur pour Markdown écrit en Nit, ainsi que sa bibliothèque `markdown`, pour l'analyse des fichiers et commentaires écrits en Markdown.

Notre spécification propose une extension au langage Markdown, l'extension `doc_commands`, permettant à l'écrivain d'introduire des directives de documentation dans ses commentaires afin de faciliter la synchronisation de l'information (section 5.4). Nous démontrons leur utilisation avec `nitiwiki`, un générateur de sites Wiki basé sur Markdown et les directives de documentation. Ce générateur est utilisé pour le site officiel du langage Nit.

Les commandes facilitent la création de références vers les entités de l'API. Ces références seront utilisées plus tard pour la création de liens vers les pages produites par les outils d'auto-documentation d'API Nit (section 5.5). Elles encouragent aussi l'utilisation d'abstractions grâce aux directives de documentation en permettant à l'écrivain d'introduire des listes et des diagrammes UML directement dans ses commentaires (section 5.6) avec l'outil `nituml`, un générateur de diagrammes UML basé sur le code.

5.1 Uniformiser la structure des projets avec `nitpackage`

Le premier objectif de notre spécification est de permettre à un développeur prenant en main un nouveau projet de s'y retrouver plus rapidement. Une solution possible pour régler ce problème est d'uniformiser la structure des projets Nit.

Tableau 5.1: Fichiers et répertoires d'un projet selon la spécification de la documentation Nit.

`src`	Répertoire des fichiers sources Nit.
`package.ini`	Fichier de description du projet.
`man`	Répertoire des manpages des exécutables du projet.
`examples`	Répertoire des exemples d'utilisation de la bibliothèque ou du projet.
`tests`	Répertoire des tests unitaires.
`README.md`	Fichier Markdown de la documentation README du projet.

La fondation Apache, qui gère et maintient de nombreux projets libres, a déjà été confrontée à ce problème. Selon la fondation Apache, suivre une hiérarchie commune à tous les projets permet aux développeurs de se repérer plus facilement et rapidement dans un nouveau projet utilisant la même hiérarchie. Ainsi, l'outil de gestion des projets Maven, utilisé pour les projets Java de la fondation Apache, suggère une structure unifiée (The Apache Software Foundation, 2002). Nous nous sommes inspiré de cette structure et l'avons adaptée aux besoins du langage Nit.

En Nit, un *package* représente une bibliothèque ou un exécutable. Le package est une unité de distribution — au même titre que les *Eggs* de Python ou les *Gems* de Ruby — et représente donc le projet dont on souhaite unifier la structure. Le tableau 5.1 présente la structure attendue d'un package Nit selon notre spécification.

5.1.1 `src` – Répertoire des sources Nit

La structure des sources est dictée par la spécification du langage Nit et il faut s'y conformer pour préserver la rétro-compatibilité du code. Selon la spécification Nit, les sources d'un package peuvent être rangées de deux façons :

1. Directement à la racine du répertoire du package ;
2. Dans un répertoire nommé `src`.

Le choix est laissé au développeur en fonction de la taille du projet.

Les sources peuvent aussi être regroupées en des unités de rangement (des répertoires) appelés *groupes*. Le groupe contient lui-même des *modules* qui représentent les unités de compilation du langage, au même titre que les classes Java par exemple.

5.1.2 `package.ini` – Fichier de description du package

Comme nous l'avons vu dans le chapitre précédent (chap. 3), l'utilisation d'un fichier de description de package est une approche répandue. Par exemple, le gestionnaire de paquets npm pour NodeJS stipule qu'un projet doit contenir un fichier de description au format JSON nommé `package.json`. Ce fichier réunit les données importantes sur un projet et est utilisé par plusieurs outils de documentation pour extraire les informations du projet et maintenir la documentation à jour.

En Nit, afin qu'un répertoire soit reconnu par les outils comme un package, il doit contenir un fichier de description de package nommé `package.ini`. Le fichier de description de package est écrit au format INI (The Free Software Foundation, Inc., 2005) et permet de spécifier les méta-données d'un package.

L'extrait de code 5.1 montre un exemple de fichier de description pour le package `markdown`, un compilateur Markdown écrit en Nit. Le détail des clés utilisées dans le fichier et leur signification est donné dans le tableau 5.2.

Ces méta-données sont extraites par les outils de documentation Nit afin de les afficher dans la documentation d'API des packages.

```
[package]
name=markdown
desc=A Markdown parser for Nit
tags=format,lib
maintainer=Alexandre Terrasa <alexandre@moz-code.org>
license=Apache-2.0
more_contributors=Jean Privat <jean@pryen.org>, Jean-Christophe
    Beaupré <jcbrinfo@users.noreply.github.com>
[upstream]
browse=https://github.com/nitlang/nit/tree/master/lib/markdown
git=https://github.com/nitlang/nit.git
git.directory=lib/markdown
homepage=http://nitlanguage.org
issues=https://github.com/nitlang/nit/issues
```

Listing 5.1: Exemple de fichier `package.ini` pour le package `markdown`.

Tableau 5.2: Détail des clés utilisées dans un fichier `package.ini`.

`package.name`	Nom du package à afficher.
`package.desc`	Description courte.
`package.tags`	Mots-clés à associer au package.
`package.maintainer`	Mainteneur principal.
`package.license`	Licence pour la distribution.
`package.more_contributors`	Liste des contributeurs.
`upstream.browse`	Adresse où consulter les fichiers sources.
`upstream.git`	Adresse du dépôt Git d'où cloner les sources.
`upstream.git.directory`	Chemin relatif vers le package depuis la racine du dépôt.
`upstream.homepage`	Page de présentation officielle.
`upstream.issues`	Adresse du gestionnaire de bogues utilisé.

5.1.3 `man` – Documentation des exécutables Nit

Lorsqu'un package fournit un exécutable, il faut le documenter, notamment, en indiquant ce qu'il fait, comment le lancer et comment utiliser ses options. Une pratique courante pour la documentation des exécutables sur les systèmes Unix est l'utilisation d'une page MAN (*manpages*) [1].

Les pages MAN peuvent être consultées grâce à la commande `man`. Elles sont écrites au format `groff`, un format de présentation de texte léger qui peut être ensuite compilé vers d'autres formats de présentation comme HTML ou LaTeX.

Nous avons choisi de réutiliser cette approche pour les exécutables Nit. Chaque programme Nit doit être accompagné d'un fichier MAN permettant de décrire le synopsis de la commande, d'expliquer son utilisation et son comportement et de lister les options de la commande et leur utilité. Le fichier MAN d'un exécutable doit porter le nom du binaire compilé suivi de l'extension `.man`. Par exemple, pour l'exécutable `nitmd`, un analyseur syntaxique de fichiers Markdown écrit en Nit, le nom du fichier MAN doit être `nitmd.man`. Les fichiers MAN doivent se trouver dans un sous-répertoire du package nommé `man`.

Au minimum, la page MAN d'un exécutable Nit doit contenir les sections suivantes :

`NAME` Nom de la commande et sa description courte.

`SYNOPSIS` Description synthétique de comment effectuer un appel.

`DESCRIPTION` Description longue de la commande et de son fonctionnement.

`OPTIONS` Liste des options acceptées.

Afin de faciliter l'écriture des fichiers MAN, Nit accepte une version Markdown de ces fichiers qui seront ensuite compilés vers groff grâce à l'outil `nitmd`, un compilateur Markdown écrit en Nit. La figure 5.2 donne un exemple de fichier MAN pour l'exécutable `nitmd` écrit en Markdown dans le fichier `nitmd.md`.

---

1. *Linux Manual pages* : `http://man7.org/linux/man-pages/man7/man-pages.7.html`

```
# NAME

nitmd - CLI for the Nit Markdown parser

# SYNOPSIS

nitmd [-t format] [files.md...]

# DESCRIPTION

`nitmd` is a Markdown parser for Nit.
It takes the path to a Markdown file as argument and compiles
it in another presentation format depending on the output
format requested.

# OPTIONS

### `-t`, `--to`
Specify output format (`html`, `md`, `man`, `latex`)

### `--metrics`
Show metrics about Markdown syntax in input files

### `-h`, `-?`, `--help`
Show Help (the list of options).
```

Listing 5.2: Exemple de fichier MAN `nitmd.md` pour l'exécutable `nitmd`.

```
# Customizing the Markdown to HTML rendering behavior
module example_custom_html is example

import markdown_html_rendering

# We customize the output for Markdown paragraphs
redef class MdParagraph
    # We customize the HTML to add a custom CSS class
    redef fun render_html(v) do
        v.add_html "<p class=\"my-css-class\">"
        visit_all(v) # Visit all children
        v.add_html "</p>"
    end
end
```

Listing 5.3: Exemple d'utilisation de l'annotation `is example` sur une méthode de la bibliothèque `markdown`.

5.1.4 `examples` – Exemples d'utilisation exécutables

Comme nous l'avons montré, les développeurs travaillant sur des bibliothèques ou des exécutables nécessitent des exemples d'utilisation (section 1.1). Nous avons donc intégré la gestion des exemples dans notre spécification.

Puisque les développeurs préfèrent les exemples qui compilent et s'exécutent, nous avons choisi de matérialiser les exemples d'utilisation sous la forme de modules Nit — rappelons que les modules représentent l'unité de compilation Nit.

Pour être reconnus comme tels par les outils de documentation, les exemples doivent être identifiés grâce à l'annotation `is example`, ce qui permet de séparer le code d'exemple du code de l'API. L'extrait de code 5.3 montre comment utiliser cette annotation pour présenter un exemple d'utilisation de la bibliothèque `markdown`.

```
module test_custom_html is test

import example_custom_html

class TestHtmlRendering
    test

    fun test_render_paragraph is test do
        var md = "some text"
        var html = "<p class=\"my-css-class\">some text</p>"
        assert md.md_to_html == html
    end
end
```

Listing 5.4: Exemple de test unitaire NitUnit pour la bibliothèque `markdown`.

5.1.5 `tests` – Tests unitaires

Selon notre spécification, les tests unitaires sont des modules Nit annotés par `is test`. Ils sont exécutés par l'outil `nitunit`. Cet outil, similaire à `JUnit`[2] du langage Java, permet de vérifier le bon comportement d'une classe ou d'une méthode à l'aide d'assertions.

Un module de test peut contenir plusieurs classes regroupant des méthodes représentant les tests unitaires. Les classes de tests à exécuter par `nitunit` sont identifiées à l'aide de l'annotation `test` — comme c'est le cas pour la classe `TestHtmlRendering` présentée dans l'extrait de code 5.4 — alors que les tests unitaires sont décrits par des méthodes annotées par `is test`.

Les tests unitaires permettent ainsi de tester le code mais aussi les exemples, tel qu'illustré par la relation entre les exemples de code 5.3 et 5.4.

---

2. `http://junit.org/junit5/`

5.1.6 `nitpackage` un outil pour uniformiser la structure des projets

Nous proposons l'outil `nitpackage` pour aider à produire et maintenir la structure uniforme des projets Nit, et ce par l'intermédiaire de la ligne de commande.

Tout d'abord, l'option `--scaffold` permet de générer la hiérarchie des répertoires par défaut et le squelette des fichiers de méta-données attendus pour les nouveaux packages.

Ensuite, il est aussi possible de transformer automatiquement la structure d'un projet existant pour l'adapter à celle préconisée dans la spécification grâce à l'option `--expand`. Cette option déplace les fichiers de sources et de méta-données existants vers les emplacements préconisés.

Mis à part la structure, `nitreadme` peut aussi générer et maintenir les fichiers de méta-données. Par exemple, l'option `--gen-ini` permet de créer le squelette du fichier de description de package et de le pré-remplir à l'aide des informations collectées depuis le code source ou l'historique de versions Git. L'option `--check-ini` vérifie la synchronisation du fichier `package.ini` ainsi créé avec les informations sur lesquelles sa génération est basée.

Les options `--gen-makefile` et `--gen-man` permettent respectivement de créer le squelette des fichiers Makefile et les pages MAN des exécutables. La tenue à jour de ces fichiers avec leurs données sous-jacentes — provenant du code — se fait grâce aux options `--check-makefile` et `--check-man`.

En plus de la création et vérification de la structure et des méta-données, `nitreadme` peut être utilisé comme crochet (*hook*) d'assurance-qualité lancé automatiquement lorsque la base de code change grâce aux *Git Hooks*[3]. Il est ainsi possible de vérifier automatiquement si les projets se conforment aux standards Nit.

---

3. `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`

Au final, l'outil `nitpackage` est utile à toutes les étapes du développement d'un projet. Il supporte le développeur dans ses tâches quotidiennes tout en l'aidant à respecter la spécification que nous proposons.

5.2 Documenter les entités Nit avec `nitmd`

Les entités du code Nit sont documentées à l'aide de deux mécanismes :

1. Les commentaires trouvés au dessus des entités du code telles les modules, classes et propriétés. En Nit, les commentaires sont préfixés du symbole `#`. L'extrait de code 5.3 (p. 131) présente un exemple de documentation appliquée à un module, une classe et une propriété.
2. Les fichiers `README.md` trouvés dans le répertoire racine des packages et des groupes. L'extrait de code 1.1 (p. 9) donne un exemple de fichier de documentation pour le package `markdown`.

Comme nous l'avons vu dans le chapitre 2, les écrivains de documentation préfèrent l'utilisation de Markdown pour la rédaction de leurs fichiers README. En effet, Markdown grâce à ses niveaux d'en-têtes permet de structurer la documentation. La présentation des blocs de code permet l'ajout d'exemples directement dans la documentation. De plus, la syntaxe des listes Markdown et les possibilités hypertexte du langage en font un bon candidat pour la présentation d'abstractions. Nous avons donc choisi d'utiliser ce format de présentation pour les commentaires Nit et les fichiers README.

Afin d'interpréter le contenu des extraits de documentation Markdown dans les outils de documentation du langage Nit, il nous faut un compilateur Markdown. La bibliothèque `markdown`[4], développée durant cette thèse, introduit un tel compilateur. Nous avons préféré l'implémentation d'un compilateur natif en Nit plutôt que

4. `https://github.com/nitlang/nit/tree/master/lib/markdown`

l'utilisation d'un compilateur déjà existant comme `pandoc` (MacFarlane, 2006) afin de limiter les dépendances du projet Nit et faciliter l'intégration du compilateur Markdown avec les outils de documentation Nit.

L'outil `nitmd`[5] — introduit dans le package `markdown` — permet de compiler, depuis la ligne de commande, des documents Markdown vers un langage de présentation tel que HTML (option `-t html`), Markdown (`-t md`)[6], groff (`-t man`) ou LaTeX(`-t latex`).

Depuis sa création initiale par Gruber (2004), Markdown a vu la création de nombreux compilateurs. Aujourd'hui, GitHub liste plus de 4000 entrées pour le label `markdown`[7]. Toutes ces implémentations différentes ont apporté leur lot d'extensions et d'exceptions autour de la spécification originale du format.

Devant tant de popularité et de variations, l'initiative CommonMark (MacFarlane *et al.*, 2004) a tenté d'en définir une spécification commune et non-ambiguë. Cette spécification propose une suite de 627 tests permettant de vérifier la validité d'une compilation vers HTML depuis Markdown. Cette suite de tests peut être exécutée sur n'importe quel compilateur Markdown avec l'outil fourni par CommonMark :

```
python3 test/spec_tests.py --program <compilateur à tester>
```

Afin de vérifier la compatibilité du compilateur `nitmd`, nous l'avons vérifié avec cette suite de tests et nous avons comparé les résultats avec `pandoc` — un outil de conversion populaire sur les systèmes Unix — et avec `nitmd-old` — l'ancien compilateur Markdown pour Nit développé avant cette thèse. Les résultats obtenus sont présentés dans le tableau 5.3

---

5. `https://github.com/nitlang/nit/tree/master/lib/markdown/nitmd.nit`

6. Notamment, pour du *pretty printing*.

7. `https://github.com/topics/markdown`

Tableau 5.3: Résultats de l'exécution de trois outils sur les suites de tests de la spécification CommonMark.

Outil	Tests réussis	Tests échoués
`pandoc`	293	334
`nitmd-old`	273	354
`nitmd`	**617**	**10**

Les 10 tests échoués par `nitmd` portent tous sur l'utilisation d'Unicode — une mise en œuvre en Nit d'Unicode a été faite par Bajolet (2016) mais semble-t-il de façon pas tout à fait complète.

`nitmd` respecte donc la spécification Markdown telle que proposée par CommonMark, nettement mieux que les deux autres outils. En fait, notre implémentation nous a même permis de déceler deux erreurs dans la suite de tests `spec_tests.py` qui ne correspondent pas avec la spécification officielle de CommonMark. Ainsi, les tests 162 et 467 vérifient que le compilateur Markdown conserve correctement les espaces codés (`%20`) dans les adresses des liens et des images. Or, le code Markdown fourni en entrée au compilateur ne contient pas d'espaces codés dans ces deux cas. Il semble que l'erreur se trouve dans l'outil Python utilisé pour générer la suite de tests à partir du document de spécification.

5.3 Encourager l'écriture d'exemples avec `nitunit`

Les commentaires des différentes entités du code ainsi que les fichiers README peuvent contenir des extraits de code afin de présenter des exemples d'utilisation ou pour expliquer comment utiliser les différentes fonctionnalités d'une API. Cette approche est analogue aux `doctests` de Python[8].

8. `http://docs.python.org/2/library/doctest.html`

```
# Translate a Markdown AST into a HTML string
#
# Usage:
# ~~~
# var parser = new MdParser
# var ast = parser.parse("**Hello World!**")
# var renderer = new HtmlRenderer
# var html = renderer.render(ast)
# assert html == "<p><strong>Hello World!</strong></p>\n"
# ~~~
#
# SEE: `String::md_to_html` for a shortcut.
class HtmlRenderer super MdRenderer
# ...
```

Listing 5.5: Exemple de `DocUnit` pour la classe `HtmlRenderer`.

Tel que présenté dans l'extrait de code 5.5, le cartouche de commentaire de la classe `HtmlRenderer` contient un exemple d'utilisation.

Ces exemples de code directement dans la documentation sont nommés *DocUnits* et sont testés à l'aide de l'outil `nitunit` afin de vérifier qu'ils sont syntaxiquement corrects et à jour avec le code source.

Nous combinons ainsi l'outil de lancement des tests unitaires avec celui de vérification des extraits de code directement inclus dans les commentaires du code source ou dans les fichiers README.

5.4 Synchroniser l'information avec les directives `nitiwiki`

Maintenir les informations trouvées au sein de la documentation à jour avec le code n'est pas une tâche aisée (section 1.1). Notre spécification de documentation

permet déjà de limiter la désynchronisation des exemples grâce aux DocUnits (section 5.3) mais qu'en est-il pour le reste des informations ? Par exemple pour un lien vers la documentation d'API, une liste de propriétés, un extrait de code ou un diagramme UML ?

Une solution pour lutter contre la désynchronisation du contenu de la documentation par rapport au code source est l'utilisation de directives de documentation, comme nous avons pu l'observer dans l'outil de génération de fichiers README Rebecca (section 3.1).

En effet, les directives étant compilées au moment de la génération de la documentation, celles-ci permettent de produire un résultat toujours à jour avec le code, ou sinon de signaler des avertissements si elles font référence à des entités inexistantes.

Notre solution permet donc aux écrivains d'utiliser ce genre de directives pour rédiger leur documentation. Ces directives sont implémentées à l'aide des *liens Wiki* supportés par `nitmd`. Il est ainsi possible d'insérer des directives de documentation directement dans le contenu Markdown à l'aide de la syntaxe `[[directive]]`.

Cette approche, en plus de limiter la désynchronisation, permet aussi d'alléger le travail de l'écrivain qui peut utiliser des directives de documentation au lieu de rédiger le texte à la main.

L'extrait de code 5.6 donne un exemple de fichier README modifié pour profiter des directives de documentation. Ces directives seront utilisées par les outils de documentation du langage Nit, présentés dans le prochain chapitre (chap. 6). Nous décrivons plus en détails ces directives dans les sous-sections suivantes ainsi qu'à la section 6.4. La liste complète des directives de documentation supportées par la spécification `doc_down` est disponible en annexe B.

Ces mêmes directives peuvent aussi être utilisées pour manipuler la structure des pages de documentation à produire. Nous démontrons cette fonctionnalité en

```
# [[markdown]] - [[ini-desc: markdown]]

`markdown` is a Markdown parser for Nit that follows the
CommonMark specification.

Contents:

[[toc: markdown]]

## Getting Started

These instructions will get you a copy of the project up and
running on your local machine.

### Dependencies

This project requires the following packages:

[[parents: markdown]]

### Run `nitmd`

Compile `nitmd` with the following command:

[[main-compile: markdown::nitmd]]

Then run it with:

[[main-run: markdown::nitmd]]

Options:

[[main-opts: markdown::nitmd]]

...
```

Listing 5.6: Contenu Markdown du fichier README du projet `markdown` utilisant les directives de documentation Nit.

implémentant `nitiwiki`, un moteur de wiki permettant de générer un site web au format HTML statique à partir de fichiers sources au format Markdown. Il s'agit d'un clone du moteur `ikiwiki` (écrit en Perl[9]).

`nitiwiki` étend l'outil `nitmd` pour générer un ensemble de pages HTML depuis une hiérarchie de répertoires contenant des fichiers Markdown. Il utilise les directives permettant de structurer la documentation telles que présentées en annexe (tableau B.2). Sa documentation complète est disponible à l'adresse `http://moz-code.org/nitiwiki/index.html`. Il est utilisé pour générer la version HTML du site officiel du langage Nit[10] ainsi que ma page personnelle pour les cours que j'enseigne à l'UQAM[11].

5.5 Faciliter les références aux entités de l'API avec `nitindex`

Comme nous avons pu l'observer dans notre étude des pratiques courantes de documentation dans le chapitre 2, les écrivains ont besoin de faire référence aux entités du code dans leur documentation. L'utilisation des mises en relief de code Markdown est utilisée à cet effet.

Nous avons choisi d'encourager cette pratique dans la documentation Nit en remplaçant automatiquement une entité du code trouvée entre apostrophes inverses, comme `` `MaClasse` ``, par un lien vers la documentation d'API de cette entité.

Une seconde syntaxe possible pour la création des liens vers la documentation d'API est grâce à la syntaxe des directives de documentation pour Nit. Ainsi, la mise en relief `` `MaClasse` `` est strictement équivalente à la directive `[[MaClasse]]`.

---

9. `http://ikiwiki.info/`

10. `http://nitlanguage.org`

11. `http://info.uqam.ca/~terrasa`

Nous proposons trois mécanismes pour faire référence à une entité de l'API via une mise en relief de code :

— Les **références simples** permettent à l'écrivain de référer à une entité via son nom court. Par exemple, pour référer à la classe `MaClasse`, l'écrivain peut simplement écrire `‘MaClasse‘`.

Une telle référence est simple à écrire mais entraîne un risque de conflit de noms : en effet, il est possible que le code contienne plusieurs entités nommées `MaClasse`, auquel cas il est nécessaire de désambiguïser le nom.

— Les **références canoniques** sont utilisées pour désambiguïser les références vers les entités à l'aide de leurs noms canoniques. Ainsi, en écrivant `‘monpackage::MaClasse‘`, l'écrivain peut référer à la classe `MaClasse` introduite dans `monpackage`, et ce sans risque de conflit.

— Les **références partielles** offrent une alternative aux références canoniques en ne précisant qu'une partie du nom canonique de l'entité que l'écrivain souhaite référencer. Par exemple, pour référer à la propriété `foo` introduite par `MaClasse`, l'écrivain peut écrire `‘MaClasse::foo‘`.

Cette syntaxe permet de résoudre les conflits de noms pour toute propriété `foo` n'étant pas introduite dans la classe `MaClasse`. Si `MaClasse` crée un conflit, il faudra alors avoir recours au nom canonique.

Afin de limiter la désynchronisation des commentaires avec le code, ces références sont vérifiées lors de la génération de la documentation. Pour chaque référence trouvée (commentaires ou fichiers README), on vérifie qu'une entité portant effectivement ce nom se trouve dans le code source documenté. Dans le cas contraire, les outils de documentation affichent un avertissement indiquant les références qui ne correspondent à aucune entité.

Bien sûr, toutes les utilisations des mises en relief de code Markdown ne sont pas forcément liées à des entités du code Nit. Comme nous l'avons montré dans notre

étude des fichiers README, certaines mises en relief de code sont utilisées pour présenter des numéros de version, des littéraux, et de nombreuses autres choses. Une heuristique basée sur le format des identifiants Nit est utilisée afin de limiter le nombre d'avertissements. Ainsi, si la référence mise en relief contient des espaces, des opérateurs ou seulement des chiffres, elle est automatiquement ignorée et ne produit aucun avertissement.

Des avertissements sont aussi affichés pour les références produisant des conflits. Les outils de documentation Nit listent alors quelles sont les entités en conflit pour le nom référencé et suggèrent une référence canonique appropriée. Là aussi afin de limiter le nombre d'avertissements, une heuristique basée sur le contexte du commentaire (ou du fichier README) est utilisée. Ainsi, une référence simple à la classe `MaClasse` faite dans le commentaire d'une entité de `monpackage` sera automatiquement désambiguïsée vers `monpackage::MaClasse` plutôt que vers une classe du même nom dans un autre package.

Afin de factoriser l'accès aux entités du modèle depuis les références simples, canoniques et partielles, nous proposons `nitindex`, un outil permettant d'indexer les entités du modèle d'un programme pour pouvoir les retrouver depuis une requête textuelle. Nous présentons cet outil comme élément de notre solution dans le chapitre 8.

5.6 Encourager l'utilisation d'abstractions avec `nituml`

Notre étude des fichiers README issus de GitHub (chap. 2) nous a permis de montrer que le type d'abstraction le plus utilisé sont les listes d'entités. Les listes sont pratiques en effet pour montrer le contenu d'une API.

Cependant, les listes présentent deux désavantages : 1) composer une liste d'entités peut être une tâche fastidieuse puisqu'il faut parcourir le code pour trouver les

entités à lister ; 2) une liste peut se désynchroniser quand une entité est ajoutée, supprimée ou renommée. Afin de simplifier le travail de l'écrivain, nous proposons l'utilisation de la directive `defs`. Elle permet d'insérer la liste des modules d'un package, la liste des classes d'un module ou encore la liste des propriétés d'une classe. Par exemple, la directive `[[defs: MaClasse]]` produit la liste des propriétés introduites par la classe `MaClasse`.

L'autre type d'abstraction généralement utilisé dans le domaine du génie logiciel sont les diagrammes UML. Pourtant, ceux-ci ne sont que peu présents dans les fichiers README, comme nous l'avons montré dans le chapitre 2.

Afin d'encourager les écrivains à utiliser ce genre d'abstractions, nous proposons une autre directive permettant d'introduire un diagramme de packages ou un diagramme de classes dans la documentation (commentaire ou fichier README).

Lorsque l'entité spécifiée dans la directive correspond à un package ou un groupe, le diagramme inséré est un diagramme de packages montrant les relations entre les groupes et les modules trouvés dans le package ; lorsque l'entité correspond à un module, la directive est remplacée par un diagramme de classes montrant les relations entre les classes introduites dans le module.

`nituml` est un outil de génération de diagrammes UML pour Nit. Il permet de créer, de façon simple, des diagrammes de packages et de classes à partir des sources d'une application ou d'une bibliothèque.

Que ce soit par son interface en ligne de commande ou son API, `nituml` est hautement configurable pour s'adapter aux différents besoins des utilisateurs ou des programmes clients. Il propose plusieurs types de diagrammes adaptés aux besoins du langage Nit :

packages Diagramme de packages appliqué aux packages Nit.

modules Diagramme de packages appliqué aux modules Nit.

classes Diagramme de classes appliqué aux classes Nit.

classdefs Diagramme de classes appliqué aux définitions de classes Nit en lien avec le raffinement de classes (Ducournau *et al.*, 2008).

5.7 Conclusion

Dans ce chapitre, nous avons présenté `doc_down`, notre proposition de spécification pour la documentation d'API du langage Nit. La ligne conductrice de cette spécification est la simplicité d'utilisation par l'écrivain. Nous proposons un outil, `nitpackage`, pour automatiser la transformation d'un projet selon les recommandations de notre spécification.

La spécification `doc_down` offre des mécanismes permettant de documenter toutes les entités du code. Comme dans la plupart des autres langages, les modules, classes et propriétés peuvent être documentés directement dans le code à l'aide de commentaires. En Nit, les packages peuvent être documentés à l'aide d'un fichier README et les exécutables grâce à des fichiers MAN.

Le langage Markdown, grâce aux en-têtes de sections, permet à l'écrivain de structurer son discours de manière flexible et d'adopter la structure qu'il juge nécessaire pour la bonne compréhension de sa documentation.

L'écriture des commentaires, des fichiers README pour la documentation des packages et des fichiers MAN de documentation des exécutables, se fait grâce au langage Markdown. Cette syntaxe est simple à écrire par l'utilisateur est peut être compilée dans un autre langage de présentation comme groff ou HTML.

Pour satisfaire aux besoins de notre spécification, nous avons développé l'outil `nitmd`, un compilateur Markdown pour le langage Nit. Ce compilateur respecte les standards du langage Markdown et permet d'interpréter les directives de documentation Nit.

Grâce à cette spécification, nous souhaitons aussi faciliter l'utilisation d'exemples dans la documentation, soit en annotant les exemples dans le code, soit en ajoutant les exemples directement dans le Markdown. Les exemples, tant dans le code que dans la documentation, sont vérifiés grâce aux assertions. Cette pratique vise à encourager les écrivains à produire des exemples en les réutilisant comme des tests unitaires avec `nitunit`.

Nous proposons l'utilisation des directives de documentation dans les commentaires, les fichiers README et les fichiers MAN afin de simplifier le travail de rédaction de l'écrivain et de maintenir les informations à jour avec la base de code. Nous nous basons sur ces directives pour implémenter un outil de génération de site Wiki nommé `nitiwiki`.

Les directives de documentation permettent à l'écrivain de faire référence à la documentation d'API d'une entité Nit ou d'insérer des abstractions telles que des listes et des diagrammes UML que nous générons grâce à `nituml`.

Dans le chapitre suivant (chap. 6), nous proposons une suite d'outils de documentation d'API pour le langage Nit profitant de la spécification `doc_down` et implémentés grâce au compilateur `nitmd`.

CHAPITRE VI

`DOC_TOOLS` – UN CADRE DE DÉVELOPPEMENT POUR LES OUTILS DE DOCUMENTATION NIT

Notre étude des générateurs de documentation existants, présentée chapitre 3, nous a permis d'établir l'état de l'art des fonctionnalités présentes dans de tels outils. Nous avons ainsi établi la liste des fonctionnalités jugées nécessaires pour un générateur de documentation d'API. Cette liste représente «*un idéal*», que nous souhaitons égaler.

Ainsi, nous décrivons dans le présent chapitre l'écosystème de documentation que nous avons développé pour le langage Nit. Cet écosystème, illustré à la figure 6.1, se compose de plusieurs outils liés à la documentation d'API remplissant chacun un rôle différent auprès des lecteurs et écrivains.

Les outils les plus importants sont `nitdoc` (section 6.1) le générateur de documentation d'API statique, `nitweb` (section 6.2) le serveur de documentation d'API et `nitx` (section 6.3) l'outil de documentation d'API en ligne de commande. Ces trois outils reposent sur la spécification de la documentation du langage Nit que nous avons présentée au chapitre précédent (chap. 5).

L'ensemble de ces outils nous a permis, tout au long de cette thèse, d'expérimenter différentes approches et idées qui feront partie de notre solution idéale telle que nous la présenterons dans les chapitres suivants.

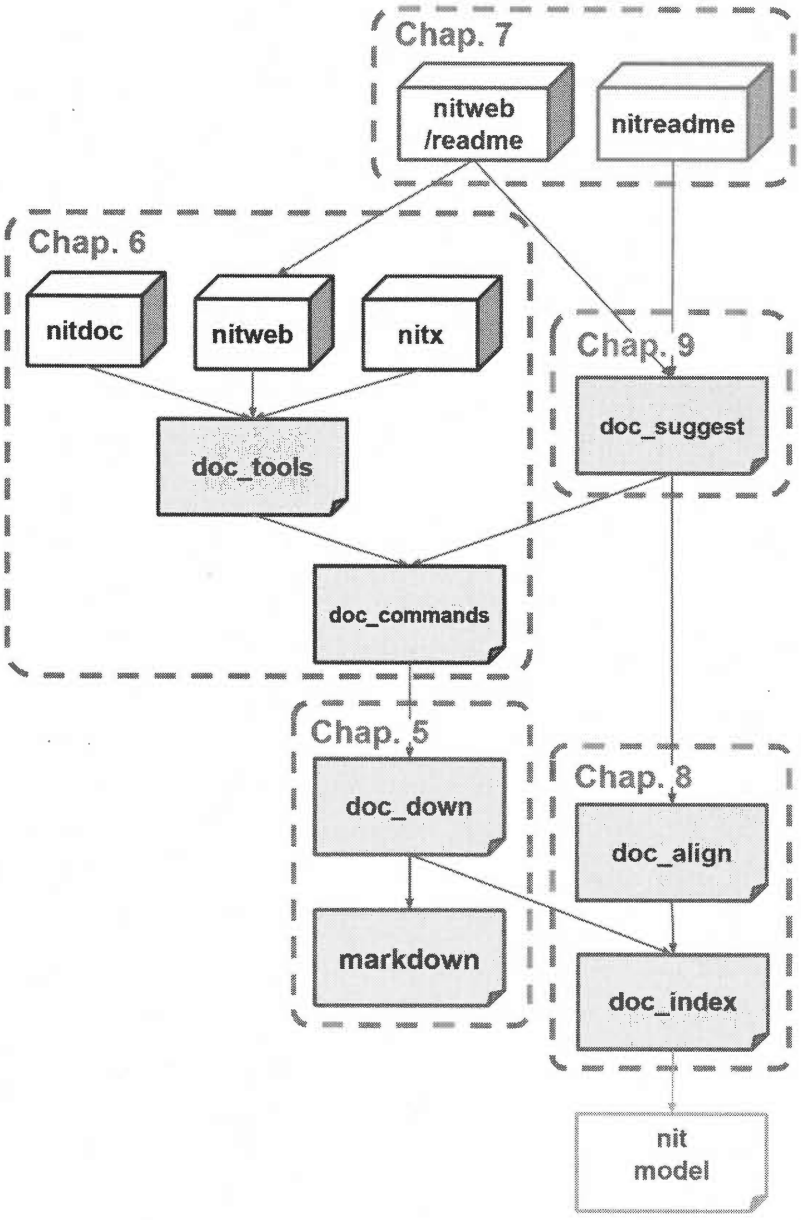


Figure 6.1: Place des outils `nitdoc`, `nitweb`, `nitx` et des modules `doc_tools` et `doc_commands` dans l'écosystème de documentation Nit.

Bon nombre de ces outils de documentation s'appuient sur des besoins communs comme l'exploration du modèle contenant les entités composant les applications et bibliothèques, à savoir les packages, modules, classes et propriétés, ou encore la génération d'extraits de documentation dans différents langages de présentation. C'est aussi le cas pour l'interprétation des directives de documentation Nit ou l'affichage des avertissements sur la documentation manquante ou périmée.

Afin de rationaliser l'effort de développement de ces outils, nous avons implémenté un cadre de développement, nommé `doc_tools`, permettant de factoriser les aspects communs à chaque outil et d'en unifier les comportements. `doc_tools` est importé par les trois outils de documentation `nitdoc`, `nitweb` et `nitx`. Il est lui-même basé sur `doc_commands`, un module qui introduit le support des directives de documentation Nit pour la documentation de l'API (section 6.4) et sur un ensemble de bibliothèques développées durant cette thèse pour supporter la création d'outils de documentation (section 6.5).

Enfin, nous comparons les fonctionnalités de notre générateur de documentation d'API `nitdoc` avec les autres outils du même type présentés au chapitre 3 (section 3.2). Nous montrons ainsi que la suite d'outils de documentation Nit égale l'état de l'art de la génération de documentation d'API (section 6.6).

6.1 `nitdoc` – Générateur de documentation d'API du langage Nit

`nitdoc` est l'outil de génération de documentation d'API du langage Nit. Au même titre que `Javadoc` (Friendly, 1995) ou `Doxygen` (van Heesch, 2008), `nitdoc` permet de créer une documentation statique au format HTML à partir du code source et des commentaires qui s'y trouvent.

`nitdoc` génère une page de documentation au format HTML pour chaque entité sélectionnée depuis le modèle. Il est ainsi possible de créer automatiquement des

pages de documentation pour l'API de chaque package, module, classe et propriété trouvé dans le code source.

Les pages HTML ainsi générées peuvent ensuite être directement ouvertes par le lecteur par l'intermédiaire d'un navigateur web ou partagées via un serveur HTTP comme on le ferait avec une documentation produite par `Javadoc`.

Le contenu de la documentation générée est basé sur l'analyse du code source et des commentaires qu'il contient. Pour chaque entité sélectionnée à partir du modèle, on affiche sa signature et son commentaire associé.

La structure de la documentation produite suit à l'identique la structure du code. En partant de la liste des packages, on peut accéder à la liste des modules contenus dans chaque package, puis à la liste des classes contenues dans chaque module, et enfin à la liste des propriétés de chaque classe.

`nitdoc` propose certaines fonctionnalités supplémentaires comparativement à un générateur de documentation traditionnel tel que `Javadoc`. Nous détaillons ici, les fonctionnalités utiles pour les lecteurs. La documentation détaillée de l'outil est disponible à l'adresse `http://nitlanguage.org/tools/nitdoc.html`.

Catalogue des packages La page d'accueil de la documentation affiche le catalogue des packages présents dans la documentation. Les packages sont triés selon leur ordre d'importance, les plus utilisés étant en haut de la liste[1]. Une capture d'écran du catalogue est proposée en annexe C.2.

1. C'est-à-dire les packages avec le plus de dépendances en premier.

Page de présentation des packages Pour chaque package, `nitdoc` génère une page de présentation contenant les informations suivantes :

- Le contenu du fichier `README.md` du package avec sa table des matières afin d'en faciliter la lecture.
- Les méta-données du package extraites du fichier de description de package (section 5.1.2) listant son mainteneur, sa licence, les liens utiles (page officielle du projet, dépôt de sources, gestionnaire de bogues), l'URL du dépôt Git permettant de cloner les sources, le nombre de commits, la date du premier et du dernier commit, la liste des mots clés, la liste des dépendances et des clients, la liste des contributeurs, des métriques donnant un aperçu de la taille du projet (nombre de modules, classes, propriétés et lignes de code).
- Un onglet avec la liste complète des groupes et modules contenus dans le package et leur documentation associée avec un lien vers la page d'API de chaque module.
- Un onglet avec le diagramme de packages montrant la place du package courant dans l'arbre d'importation des packages accompagné de la liste complète des dépendances et clients.

La figure 6.2 présente une capture d'écran de la page de présentation du package `markdown` avec les informations listées ci-dessus.

Page de présentation des entités Pour chaque module, classe et propriété, `nitdoc` génère une page de documentation contenant les informations suivantes :

- La documentation du module extraite à partir du commentaire d'en-tête de l'entité.

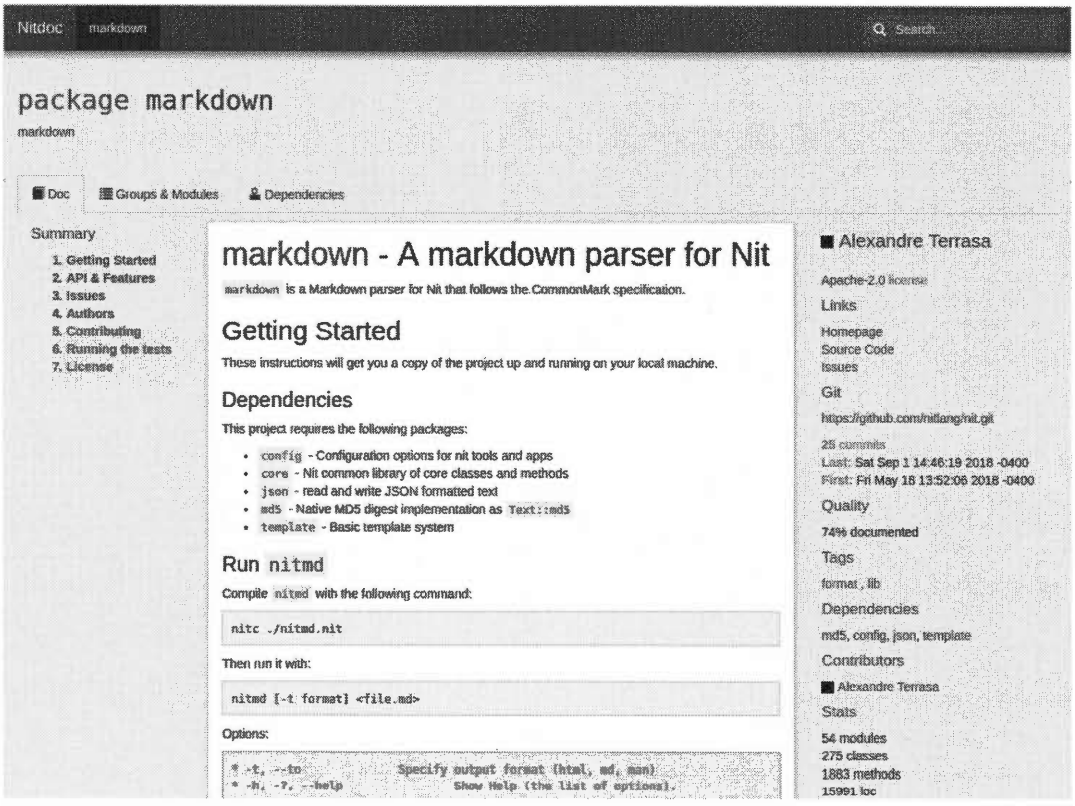


Figure 6.2: Capture d'écran de la page de présentation du package `markdown` telle que générée par `nitdoc`.

— Un onglet avec la liste complète des classes contenues dans le module et leur documentation associée avec un lien vers la page d'API de chaque classe ou un onglet avec la liste complète des propriétés contenues dans la classe et leur documentation associée avec un lien vers la page d'API de chaque propriété.

— Un onglet avec le diagramme de packages montrant la place du module courant dans l'arbre d'importation des modules accompagné de la liste complète des dépendances et clients ou un onglet avec le diagramme de classes montrant la place de la classe courante dans l'arbre d'héritage des classes accompagné de la liste complète des super-classes et sous-classes.

— Un onglet avec le code source de l'entité avec coloration syntaxique. Chaque entité du code est cliquable pour accéder à la page de documentation d'API associée. Le survol des entités permet d'afficher leur commentaire court.

— Un onglet présentant la linéarisation des définitions de la classe ou propriété courante avec le code source de chaque définition. Comme le langage Nit supporte le raffinement de classes (Ducournau *et al.*, 2008), cette vue est particulièrement utile pour comprendre l'ordre dans lequel sont appliquées les redéfinitions. La linéarisation présentée est identique à celle suivie par le compilateur (`nitc`) et l'interpréteur (`nit`) du langage.

6.2 `nitweb` – Serveur de documentation du langage Nit

`nitweb` est le serveur de documentation d'API pour le langage Nit. Il expose une API JSON/REST permettant d'explorer le modèle d'une application ou d'une bibliothèque à l'aide de requêtes HTTP.

Le serveur de documentation `nitweb` propose deux vues permettant aux clients d'accéder à la documentation d'API des entités du modèle chargé en mémoire :

Vue JSON Cette vue permet à des clients d'accéder programmatiquement à la documentation d'API des entités grâce à des requêtes HTTP de type POST ou GET. Elle est conçue pour permettre le développement d'autres outils de documentation sur le mode de fonctionnement client/serveur. Par exemple pour une application de documentation sur téléphone mobile ou tablette, d'autres applications web ou encore un outil de documentation en ligne de commande (voir `nitx` en sous-section 6.3).

Vue HTML Elle permet au lecteur d'explorer le contenu de `nitweb` par l'intermédiaire d'un navigateur web grâce à des requêtes HTTP asynchrones envoyées via JavaScript.

La vue HTML présente une interface unifiée avec la documentation d'API générée par l'outil `nitdoc`. Elle est implémenté à l'aide du framework de développement d'interface web JavaScript `Angular`[2], qui se comporte comme un client de l'API JSON/REST. Un exemple est donné en annexe C.3.

Que ce soit via l'interface JSON ou HTML, `nitweb` intègre toutes les fonctionnalités de l'outil `nitdoc`. Son aspect dynamique lui permet de proposer de nouvelles fonctionnalités utiles aux lecteurs. La documentation détaillée de l'outil se trouve à l'adresse `http://nitlanguage.org/tools/nitweb.html`.

6.3 `nitx` – Outil de documentation en ligne de commande

`nitx` est un outil de documentation d'API en ligne de commande. Il permet de naviguer au sein du modèle d'une application ou d'une bibliothèque grâce à des directives fournies sur l'entrée standard — par défaut le clavier – et d'obtenir

2. `http://angular.io/`

des informations sur les entités de ce modèle. L'utilisateur peut alors accéder à la documentation des entités sans sortir du terminal et l'outil peut être utilisé en combinaison avec d'autres outils en ligne de commande comme l'éditeur `vim`[3], éditeur particulièrement prisé par les développeurs du langage Nit[4].

`nitx` offre les mêmes fonctionnalités que l'outil `nitdoc`. Il permet également de se connecter au serveur de documentation `nitweb` pour exécuter les directives sur une machine distante et ainsi ne pas avoir à analyser le code localement.

Les fonctionnalités offertes par `nitx` sont les mêmes que celles de l'outil `nitdoc` mais par l'intermédiaire de la ligne de commande. Sa documentation détaillée se trouve à l'adresse `http://nitlanguage.org/tools/nitx.html`.

Afin de faciliter son utilisation à la fois par les utilisateurs et par les applications clientes, `nitx` offre trois modes d'exécution que nous présentons ci-dessous.

Interface interactive de commandes Dans son mode par défaut, l'outil se comporte comme un interpréteur de commandes interactif. Une fois l'outil lancé, l'utilisateur peut taper des commandes — les directives de documentation Nit — qui sont interprétées par `nitx`. Le résultat de ces directives est affiché directement dans la console grâce au format de présentation ANSI/VT100. Un exemple est donné en annexe C.1.

Exécution non-interactive de commandes Afin de faciliter son usage depuis d'autres outils en ligne de commande, l'outil `nitx` accepte l'option `--command` (ou `-c` en bref). Avec cette option, `nitx` attend une commande directement depuis le terminal et affiche son résultat sans ouvrir le mode interactif. Il est ainsi plus facile pour les outils clients de lire le résultat.

---

3. `http://www.vim.org`

4. Un *plugin* vim pour Nit est disponible à l'adresse `http://nitlanguage.org/vim/`

Connexion au serveur `nitweb` Le lancement de l'outil `nitx` nécessite un accès au code source sur lequel l'utilisateur souhaite exécuter ses directives. De plus, le lancement de l'outil peut prendre un certain temps en fonction de la taille du code source à analyser.

Pour pallier ce problème, l'option `--nitweb` permet à `nitx` de fonctionner en mode client connecté au serveur de documentation `nitweb`. Dans ce mode, les directives sont traitées à distance sur le serveur `nitweb` grâce à son API JSON/REST et les résultats sont récupérés pour être affichés dans la console de l'utilisateur.

6.4 `doc_commands` – Directives de documentation pour Nit

Afin de factoriser l'interprétation des directives de documentation Nit par ses outils, nous proposons le cadre de développement `doc_commands`. Les `doc_commands` permettent d'explorer le modèle d'une application ou d'une bibliothèque grâce à une API unifiée permettant d'en sélectionner, filtrer et organiser le contenu en vue de la construction d'une documentation d'API.

Le tableau 6.1 présente quelques-unes des directives utiles pour la création d'un générateur de documentation ou l'écriture d'un fichier README. La liste complète des directives de documentation supportées par `doc_commands` est disponible en annexe B.

Ce cadre est directement inspiré du patron de conception *Commandes* tel que décrit par Gamma *et al.* (1995). Il permet de séparer le code de création et initialisation de la directive du code de l'action elle-même, ce qui offre une plus grande flexibilité dans le type de clients utilisant ces directives.

Tableau 6.1: Directives de documentation proposées par le cadre de développement `doc_commands`.

`CmdComment`	Récupère la documentation (commentaire ou fichier README) associée à une entité du modèle.
`CmdParents`	Récupère la liste des parents (dépendances de packages, importation de modules ou super-classes) d'une entité du modèle.
`CmdFeatures`	Récupère la liste des entités imbriquées dans une entité du modèle telles que les modules dans un package, les classes dans un module ou les propriétés dans une classe.
`CmdEntityCode`	Récupère le code source associé à une entité du modèle.
`CmdExamples`	Récupère la liste des exemples associés à une entité du modèle.
`CmdUML`	Compose un diagramme UML (diagramme de packages ou de classes) pertinent pour une entité du modèle.

```
import doc_commands

# Création et paramétrage de la directive
var cmd = new CmdChildren(model,
  entity_name = "markdown::MdRenderer",
  examples = false,
  tests = false,
  limit = 3)

# Exécution de la directive
cmd.init_command

# Traitement des résultats
print "Children of `markdown::MdRenderer`:"
for child in cmd.results do
  print " * {child.full_name}"
end
```

Listing 6.1: Exemple d'utilisation de la directive de documentation `CmdChildren`.

Utilisation d'une directive en Nit L'extrait de code 6.1 illustre l'utilisation de la directive `CmdChildren` depuis son interface Nit. Cette directive récupère la liste des enfants directs d'une entité du modèle, liste qui peut être les packages qui importent un package, les modules qui importent un module ou les classes qui héritent d'une classe.

Dans cet exemple, on cherche à récupérer la liste des sous-classes de l'interface `markdown::MdRenderer` en omettant les classes d'exemples et les classes liées aux tests unitaires et en limitant le nombre de résultats à trois classes. En fonction des directives, d'autres options de configuration peuvent être spécifiées.

Une fois exécuté, l'extrait de code 6.1 affichera ce qui suit :

```
Children of `markdown::MdRenderer`:
 * markdown::HtmlRenderer
 * markdown::LatexRenderer
 * markdown::ManRenderer
```

Interfaces d'utilisation des directives de documentation Afin de satisfaire les besoins des différents outils de documentation, les directives de documentation peuvent être exécutées depuis différentes interfaces. Le tableau 6.2 dresse la liste des interfaces supportées et des clients les utilisant.

Présentation des résultats des directives de documentation La présentation des résultats d'une directive de documentation dépend du type de client qui l'utilise. Il est donc utile de pouvoir spécifier à chaque directive le type de rendu attendu pour le résultat, ce qui peut être spécifié directement à l'initialisation de la directive via l'option `format`. Les formats de rendu supportés sont présentés dans le tableau 6.3.

Tableau 6.2: Interfaces de manipulation des directives de documentation.

`JSON`	Interface JSON des directives de documentation. Cette interface est utilisée par le serveur de documentation `nitweb` avec son API JSON/REST (voir sous-section 6.2) via des requêtes HTTP/POST.
`HTTP`	Interface HTTP/GET des directives de documentation. Afin de faciliter leur utilisation via des requêtes HTTP, les directives peuvent aussi être initialisées et exécutées par l'intermédiaire de requêtes HTTP/GET. Bien que redondante avec l'interface JSON, nous fournissons l'interface HTTP/GET afin de faciliter l'utilisation des directives par l'intermédiaire de la barre d'adresse du navigateur ou via un outil en ligne de commande tel que `cURL`. Cette interface est elle aussi utilisée par le serveur de documentation `nitweb` avec son API JSON/REST (voir sous-section 6.2).
`Texte`	Interface texte des directives de documentation. Cette interface est utilisée par l'outil de documentation en ligne de commande `nitx` (sous-section 6.3). L'interface texte permet aussi l'interprétation des directives de documentation trouvées dans les commentaires et les fichiers README (sous-section 6.4) en suivant le format entre double crochets.

Tableau 6.3: Formats de rendu des directives de documentation.

`ANSI`	Pour affichage dans le terminal utilisé par l'outil de documentation en ligne de commande `nitx` (section 6.3).
`HTML`	Pour la présentation dans le navigateur et utilisé par le générateur de documentation `nitdoc` (section 6.1).
`JSON`	Pour l'échange des informations issues de la documentation et utilisé par le serveur de documentation `nitweb` (section 6.2).
`LATEX`	Pour la création de manuels PDF. Format utilisable via le serveur de documentation `nitweb` en fonction du paramétrage de la directive.
`MD`	Rendu au format Markdown utilisable via le serveur de documentation `nitweb`.
`DOT`	Rendu des figures Dot. Certaines directives, comme la directive `CmdUML`, permettent de générer des diagrammes à l'aide de l'outil `dot`. Elles supportent alors les formats `png`, `svg`, `pdf` et `dot`.
`CODE`	Rendu du code source. Certaines directives, comme `CmdEntityCode`, permettent de retourner du code source. Elles supportent alors les formats `html`, `ansi`, `latex` et `nit`.

Tableau 6.4: Bibliothèques additionnelles développées dans le cadre de cette thèse.

`config`	Bibliothèque de gestion des options de configuration, utilisée pour la configuration des outils de documentation.
`console`	Bibliothèque de manipulation du terminal et d'affichage au format ANSI, utilisée par l'outil de documentation en ligne de commande `nitx`.
`dot`	Bibliothèque de création de graphiques au format DOT et de génération d'images avec `graphviz`[5], utilisée par `nituml` pour la création des diagrammes de packages et de classes.
`github`	Connecteur vers l'API JSON/REST de GitHub[6], utilisé pour permettre la connexion des utilisateurs sur le serveur `nitweb` et pour la récupération des informations du dépôt Git des projets à documenter.
`html`	Bibliothèque de génération de contenu HTML, utilisée pour créer le contenu HTML produit par les outils de documentation `nitdoc` et `nitweb`.
`ini`	Analyseur syntaxique de fichiers au format INI, utilisé pour lire les fichiers de description des packages et pour la configuration des outils de documentation.
`mongodb`	Connecteur de base de données vers MongoDB[7], utilisé par le serveur de documentation `nitweb` pour le stockage des données utilisateurs.
`nlp`	Bibliothèque d'analyse de la langue naturelle (*Natural Language Processing*), utilisée pour l'analyse des commentaires en anglais et l'implémentation du moteur de recherche plein texte de `nitweb` ; utilise la bibliothèque StanfordNLP pour l'analyse syntaxique de l'anglais et la lemmatisation des termes utilisés dans les commentaires[8].
`popcorn`	Bibliothèque de développement d'API REST pour Nit, utilisée par le serveur de documentation `nitweb` pour exposer l'API JSON/REST permettant d'accéder à la documentation des entités du modèle et des fonctionnalités connexes.
`saf`	*Nit Static Analysis Framework*, un cadre de développement pour l'analyse statique du code source Nit, utilisé pour l'analyse des extraits de code et l'extraction d'exemples.
`vsm`	Bibliothèque de modélisation vectorielle, utilisée pour l'implémentation du moteur de recherche plein texte de `nitweb`.

6.5 Bibliothèques complémentaires pour les outils de documentation

Tout au long du travail ayant mené à cette thèse, de nombreuses bibliothèques Nit ont été conçues et mises en œuvre afin de supporter la création des outils de documentation présentés précédemment. Nous fournissons dans le tableau 6.4 la liste des bibliothèques les plus importantes.

Au final, et selon la commande `git log --author` appliquée sur le dépôt Git du langage Nit, ce travail représente un ensemble de 3 109 commits et 473 581 lignes de code.

6.6 Comparaison avec les autres générateurs de documentation d'API

L'ensemble des fonctionnalités des outils `nitdoc`, `nitweb` et `nitx` représente l'*idéal* que nous cherchions à atteindre par rapport aux outils de documentation d'API des autres langages tels que nous les avons présentés dans le chapitre précédent (chap. 3). Le tableau 6.5 présente cet idéal pour Nit en comparaison avec les autres outils étudiés. Comme on peut le remarquer, toutes les dimensions sont couvertes.

L'utilisation de la page de présentation des packages et des groupes sous la forme d'un fichier README au format Markdown permet à l'écrivain de structurer simplement sa documentation à l'aide des en-têtes de sections et des paragraphes. Les exemples sont ajoutés directement dans le commentaire des entités Nit comme

---

5. `http://www.graphviz.org/`

6. `http://developer.github.com/v3/`

7. `http://www.mongodb.com/`

8. `https://nlp.stanford.edu/`

Tableau 6.5: Comparaison des fonctionnalités offertes par `nitdoc` par rapport aux autres générateurs de documentation d'API présentés au chapitre 3.

	Structure			Exemples			Abs.		Qualité	
Outils	Sélection	Regroupement	Ordre	Présentation	Réification	Vérification	Listes	Figures	Style	Synchro.
nitdoc	•	•	•	•	•	•	•	•	•	•
Doxygen	•	•	•	•	•		•	•	•	•
Sphinx	•	•	•	•	•	•	•			
Scribble	•	•	•	•	•	•				•
ScalaDoc				•	•		•	•	•	•
SandCastle	•	•	•	•	•					•
YARD			•	•	•			•	•	•
Godoc				•	•	•		•		
YUIDoc			•	•	•		•			
Haddock	•	•		•			•			
phpDocumentor	•	•	•	•				•		
Perldoc			•	•					•	•
PhD	•	•	•	•						
Javadoc			•						•	•
ApiGen			•	•						
JSDoc			•	•	•					
Universal Report				•				•		
Docco				•						
RDoc				•						
Doxx				•						
Autodoc				•						
Codox				•						
AppleDoc										
Natural Docs										

des blocs de code Markdown et sont affichés par `nitdoc` (ou `nitweb` ou `nitx`) lors de la génération de la documentation de l'entité. Ils sont vérifiés grâce à l'outil de tests unitaires `nitunit`. Les abstractions telles les listes et les diagrammes UML sont insérées directement dans le fichier README à l'aide des directives de documentation. Elles sont aussi affichées par défaut dans `nitdoc` pour les packages, les groupes, les modules et les classes. Les commentaires et fichiers README sont analysés par `doc_commands` pour vérifier le contenu et l'utilisation des directives de documentation puis par `doc_tools` afin d'en vérifier le style.

6.7 Conclusion

Dans ce chapitre, nous avons présenté l'écosystème de documentation d'API pour le langage Nit en décrivant les outils développés pour générer cette documentation ainsi que le cadre de développement mis en place pour faciliter la création de ces outils.

Nous avons tout d'abord introduit les trois outils permettant de générer la documentation d'API en Nit développés durant cette thèse. **1.** `nitdoc` permet de générer une documentation d'API statique composée de fichiers HTML visualisable grâce à un navigateur à l'instar de `Javadoc`. **2.** `nitweb` offre une vue dynamique sur la documentation, qui peut être utilisé par l'intermédiaire de son API REST par d'autres outils ou par le lecteur via son interface HTML dans un navigateur. **3.** `nitx` offre lui aussi une vue dynamique de la documentation d'API mais cette fois-ci via la ligne de commande, qui peut ainsi être utilisé par d'autres outils ou directement par le lecteur.

L'implémentation de ces trois outils présente des points communs que nous avons tâché de factoriser grâce à un cadre de développement, nommé `doc_tools`, spécifique pour la création de générateurs de documentation d'API. `doc_tools` s'appuie

lui même sur `doc_commands` qui introduit le support des directives pour la documentation d'API. Elles peuvent être exécutées depuis différents types de clients et retournent l'information dans divers formats, tels que JSON, HTML, ANSI/VT-100 ou LaTeX.

L'écosystème présenté dans ce chapitre représente un pas de plus vers la création d'un outil de génération de fichiers README. En effet, grâce à `doc_commands`, nous pouvons maintenant produire du contenu pour les directives utilisées par l'écrivain dans son fichier README. Et grâce aux outils de documentation, nous pouvons afficher ce contenu au lecteur. Le chapitre suivant (chap. 7) présente notre solution pour un outil d'assistance à la rédaction de fichiers README capable de suggérer l'utilisation de directives de documentation Nit à l'écrivain.

Troisième partie

Rédaction assistée de fichiers

README pour Nit

CHAPITRE VII

`NITREADME` : UN OUTIL POUR ASSISTER LA RÉDACTION DE FICHIERS README

Suite à notre étude du contenu des fichiers README (chap. 2) et des générateurs existants (chap. 3), nous avons dressé le portrait d'une solution idéale que nous avons caractérisée selon quatre critères :

- Génération assistée de l'échafaudage du fichier README ;
- Génération des sections dédiées à la documentation de l'API ;
- Annotation, vérification et insertion automatique des exemples ;
- Insertion automatique des abstractions telles les listes et les figures.

Nos expérimentations sur une génération automatique des sections de documentation de l'API (chap. 4) ont montré les limites d'une solution complètement automatisée par rapport aux attentes des écrivains.

Dans le présent chapitre, nous présentons notre solution pour la génération et la maintenance de fichiers README selon une approche *semi-automatisée* mettant l'écrivain de la documentation au centre du processus de rédaction. Cette solution se matérialise sous la forme d'un outil de documentation nommé `nitreadme` qui permet *d'assister les écrivains dans la rédaction puis la maintenance de fichiers* README. Sa place dans l'écosystème de documentation est illustrée à la figure 7.1.

Nous décrivons ici notre solution selon le point de vue de l'utilisateur de l'outil, donc ses différentes fonctionnalités, en détaillant leurs rôles et comment les utiliser.

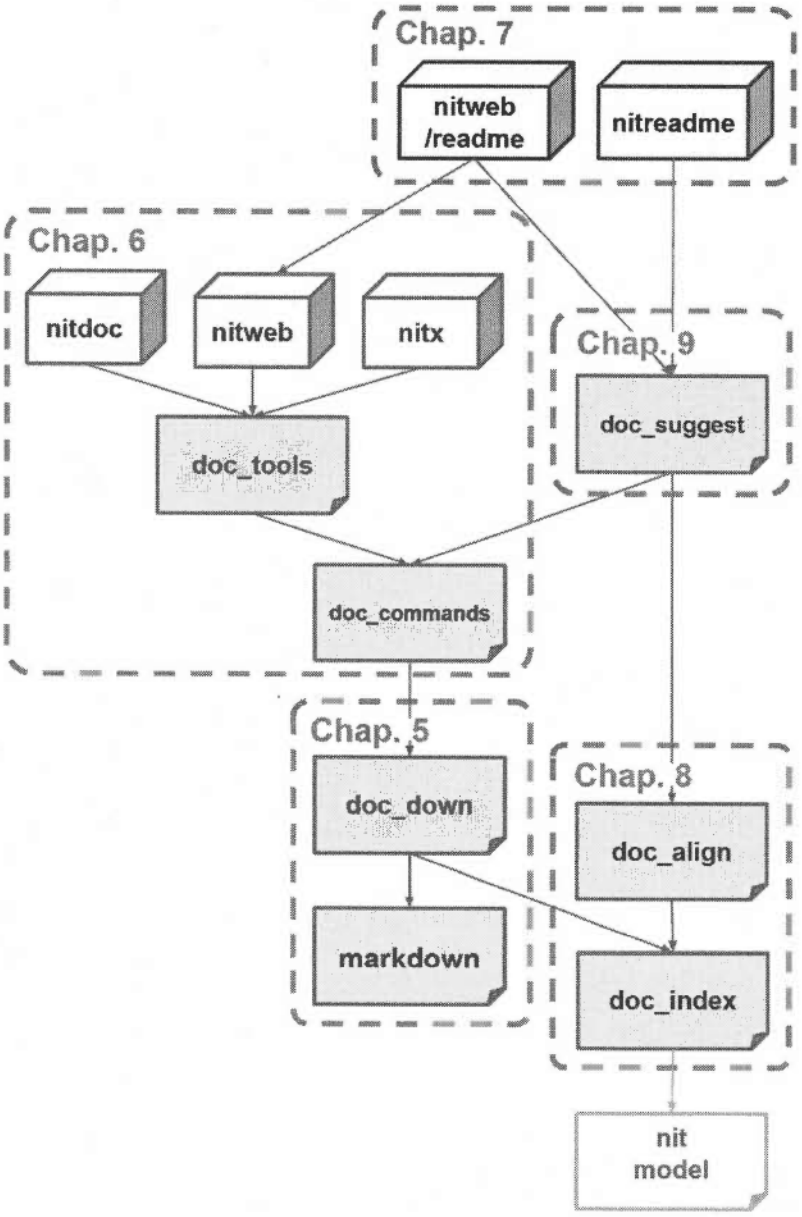


Figure 7.1: Place de l'outil `nitreadme` dans l'écosystème de documentation Nit.

Les chapitres subséquents présenteront la mise en œuvre de ces fonctionnalités et leurs rouages internes, détails d'intérêt secondaire pour un utilisateur.

L'outil `nitreadme` permet à un écrivain de documentation d'écrire, d'améliorer et de maintenir son fichier README en lui suggérant du contenu à insérer dans le fichier, tels que des liens vers la documentation d'API, l'importation de commentaires depuis le code source, l'ajout de listes de propriétés, d'exemples ou encore de diagrammes UML.

Comme l'illustre la figure 7.2, l'écrivain peut utiliser `nitreadme` par l'intermédiaire de deux interfaces : l'interface en *ligne de commande* (section 7.1) et l'interface *graphique* grâce au module d'extension pour le serveur de documentation `nitweb` (section 7.2).

Les suggestions sont basées sur les directives de documentation Nit (section 6.4) et sont choisies en fonction des observations faites sur le contenu traditionnel des fichiers README (chap. 2). Toutes ces suggestions sont fondées sur l'analyse du code source que l'écrivain est en train de documenter et se font au cours de l'écriture du fichier README. Nous présentons plus en détails ces différents types de suggestions dans la section 7.3.

7.1 `nitreadme` – Utilisation depuis la ligne de commande

Au même titre que tous les outils de la chaîne de compilation et de documentation du langage Nit, `nitreadme` peut être utilisé par une interface en ligne de commande. Cette interface permet à l'utilisateur d'interagir avec l'outil grâce à des options passées au lancement de `nitreadme`. Le fait qu'il soit utilisable comme une commande le rend compatible avec les crochets de modifications de code tels que disponibles avec Git [1] facilitant ainsi l'intégration de l'outil dans le processus de développement.

---

1. `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`

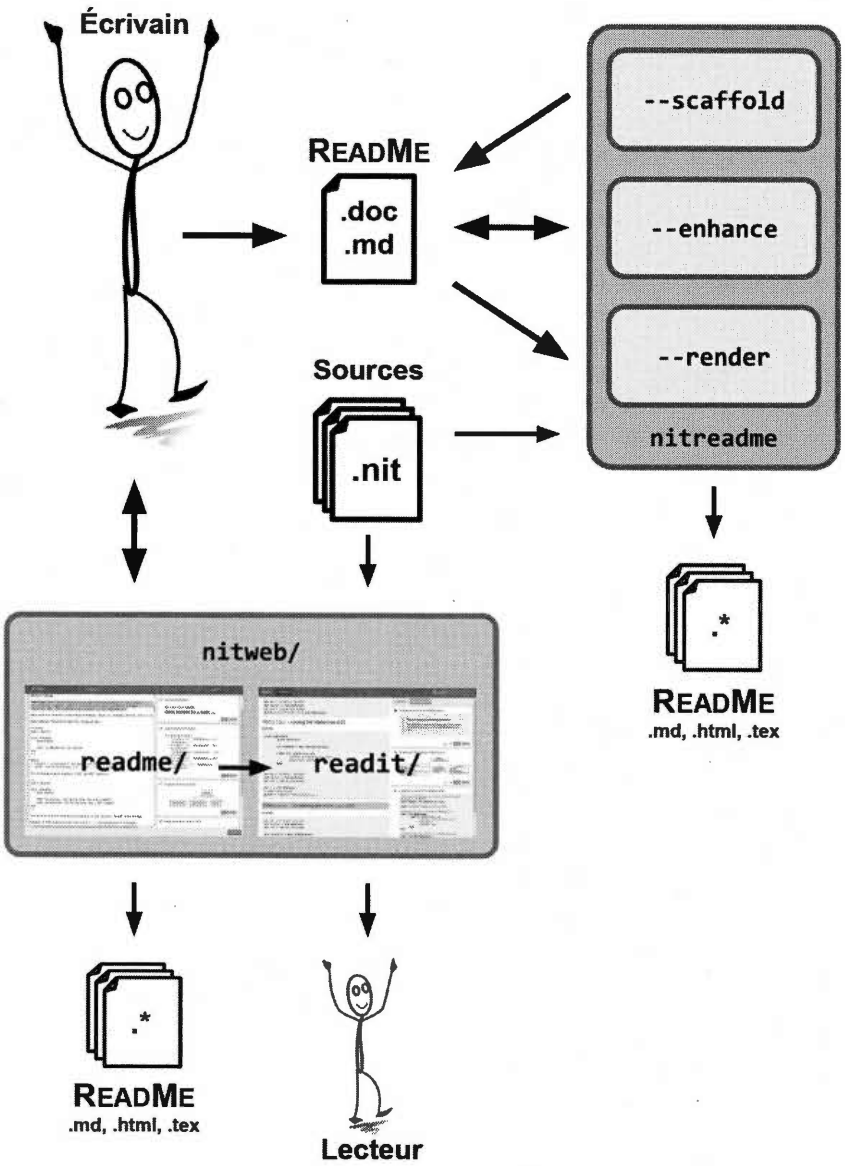


Figure 7.2: Processus d'utilisation de l'outil `nitreadme` par la ligne de commande et par le module d'extension au serveur de documentation `nitweb`.

Les sous-sections suivantes décrivent les différents modes d'exécution de la commande `nitreadme`.

7.1.1 `--scaffold` – Échafaudage d'un nouveau README

La commande `nitreadme` exécutée avec l'option `--scaffold` permet de générer le squelette d'un nouveau fichier README pour un package. Par exemple, pour générer le squelette pour le package `markdown` :

```
nitreadme --scaffold lib/markdown
```

Une fois exécutée, la commande crée un nouveau fichier Markdown nommé `README.doc.md` contenant l'échafaudage du fichier selon les cartes de suggestions présentées à la section 7.3. L'écrivain peut alors modifier le contenu du squelette à sa guise en fonction des informations qu'il souhaite ajouter. La figure 7.3 présente un extrait du squelette de fichier `README.doc.md` généré par `nitreadme` pour le package `markdown`.

Un fichier nommé `README.doc.md` correspond à un fichier de documentation intermédiaire. Il contient la documentation au format Markdown en incluant des directives de documentation. Il peut être compilé vers un fichier `README.md` afin de remplacer les directives de documentation qu'il contient par du contenu Markdown brut, plus facile à lire.

Quand le fichier `README.doc.md` est présent dans un package, les outils de documentation du langage Nit, tels `nitdoc` ou `nitweb`, l'utilisent comme fichier principal de documentation à afficher dans les pages de packages et de groupes. Si ce fichier n'existe pas, alors les outils utilisent le fichier `README.md`.

```
# [[markdown]] - [[ini-desc: markdown]]

[[toc: markdown]]

Example from `markdown::example_parsing`:

[[code: markdown::example_parsing]]

## Getting Started

These instructions will get you a copy of the project up and
   running on your local machine.

### Dependencies

This project requires the following packages:

[[parents: markdown]]

### Run `nitmd`

Compile `nitmd` with the following command:

[[main-compile: markdown::nitmd]]

Then run it with:

[[main-run: markdown::nitmd]]

Options:

[[main-opts: markdown::nitmd]]

## Features

...
```

Figure 7.3: Extrait du squelette de fichier `README.doc.md` généré par la commande `nitreadme --scaffold` pour le package `markdown`.

7.1.2 `--enhance` – Suggestion du contenu

La suggestion de contenu s'effectue sur un fichier `README.doc.md` existant. Elle permet d'introduire automatiquement des directives de documentation dans le fichier, par exemple, pour ajouter les directives de listes, d'import d'extraits de code, d'exemples ou de diagrammes UML. Les directives sont directement ajoutées dans le contenu Markdown du fichier aux emplacements les plus cohérents.

Ce processus se lance grâce à la commande suivante :

```
nitreadme --enhance lib/markdown/
```

L'écrivain peut choisir de visualiser les différences entre le fichier original et le fichier tel que modifié par `nitreadme` grâce à l'option `--diff`. Ainsi, la commande suivante génère un fichier Markdown temporaire nommé `README.suggest.md` qui peut être ensuite comparé avec le fichier `README.doc.md` original :

```
nitreadme --enhance --diff lib/markdown/
```

7.1.3 `--check` – Vérification du contenu

L'option `--check` permet à l'écrivain de lancer `nitreadme` sur le fichier source `README.doc.md` afin d'en vérifier le contenu. Avec cette option, `nitreadme` affiche des avertissements en cas de références périmées, de directives incorrectes ou de contenu manquant selon les cartes de suggestions présentées dans ce chapitre.

```
nitreadme --check lib/markdown/
```

Si un fichier `README.md` existe déjà dans le projet, alors `nitreadme` vérifie aussi la synchronisation de son contenu avec celui du fichier `README.doc.md` et alerte l'écrivain s'il est nécessaire de le compiler à nouveau dans le cas où le contenu produit par les directives a changé.

7.1.4 `--render` – Compilation du fichier README

Un inconvénient de l'utilisation de directives de documentation est que ces dernières rendent le fichier plus difficile à lire dans son état brut. Afin de parer à cet inconvénient, l'écrivain a la possibilité de compiler — de traduire — son document README dans un format de présentation plus lisible sans utilisation de l'outil.

Pour partager sa documentation, l'écrivain peut compiler le fichier intermédiaire `README.doc.md` vers le fichier `README.md` en transformant les directives de documentation Nit en du contenu Markdown brut ou d'autres formats. Par exemple, la commande suivante permet de générer le fichier README final à partir du fichier `README.doc.md` du package `markdown` :

```
nitreadme --render lib/markdown/
```

Une fois exécutée, cette commande produit le fichier `README.md` final en remplaçant les directives par le contenu Markdown qui sera présenté au lecteur. Les directives sont toutes transformées en contenu Markdown brut à l'exception des directives de diagrammes UML qui sont quant à elles transformées en des images puis incluses dans le fichier final grâce à la syntaxe d'inclusion d'images du format Markdown — voir section 2.7 pour plus d'information sur la syntaxe des images Markdown.

Le format de sortie Markdown est utilisé pour ajouter le README dans les sources du projet tel que le fichier `README.md` traditionnellement fourni dans les projets Nit. Ce format est automatiquement reconnu par les plate-formes de partage de code comme GitHub, GitLab ou BitBucket. En plus de Markdown, `nitreadme` propose la compilation vers les formats HTML, LaTeX et PDF.

Il est conseillé aux écrivains de conserver le fichier intermédiaire `README.doc.md` afin de pouvoir le modifier et le mettre à jour ultérieurement. Le fichier `README.md` devrait lui aussi être versionné afin d'être présenté aux lecteurs lorsqu'ils visualisent le projet sur une plate-forme de partage de code telle que GitHub ou GitLab.

7.2 `nitweb` – Utilisation de `nitreadme` par son interface graphique

En plus de son interface en ligne de commande, `nitreadme` peut être utilisé par l'intermédiaire d'une interface graphique web et d'un navigateur. Ce mode fait partie intégrante du serveur de documentation, nommé `nitweb`, présenté au chapitre 6.2. Une fois le serveur `nitweb` lancé, l'outil `nitreadme` est accessible à l'adresse `http://localhost:3000/readme`.

7.2.1 `nitweb/readme` – Assistant de rédaction

Lorsque l'écrivain est connecté à `nitweb/readme` via son navigateur, l'outil lui affiche l'interface présentée à la figure 7.4. La zone de texte à gauche permet à l'écrivain de saisir sa documentation au format Markdown et supporte la coloration syntaxique pour ce format. La colonne de droite contient des suggestions relatives au texte en cours d'écriture ou à la portion de texte sélectionnée.

À l'instar des suggestions d'auto-complétion de code faites dans un IDE[2], les suggestions de documentation nécessitent un espace de présentation plus grand qu'une simple liste déroulante, notamment en terme d'espace vertical. En effet, là où un menu déroulant permet de présenter une liste de méthodes pour l'auto-complétion, cette approche n'est pas idéale pour la présentation d'un exemple de code complet ou d'un diagramme UML, tous deux nécessitant plus d'espace vertical.

Afin de mieux présenter les suggestions à l'écrivain, nous réutilisons le concept de cartes utilisé par Google dans son outil `SpreadSheet Explore` (Google, 2017). `SpreadSheet Explore` affiche des cartes suggérant des analyses à appliquer sur les données sélectionnées dans la feuille de calcul. Ces analyses se matérialisent

2. *Integrated Development Environment.*

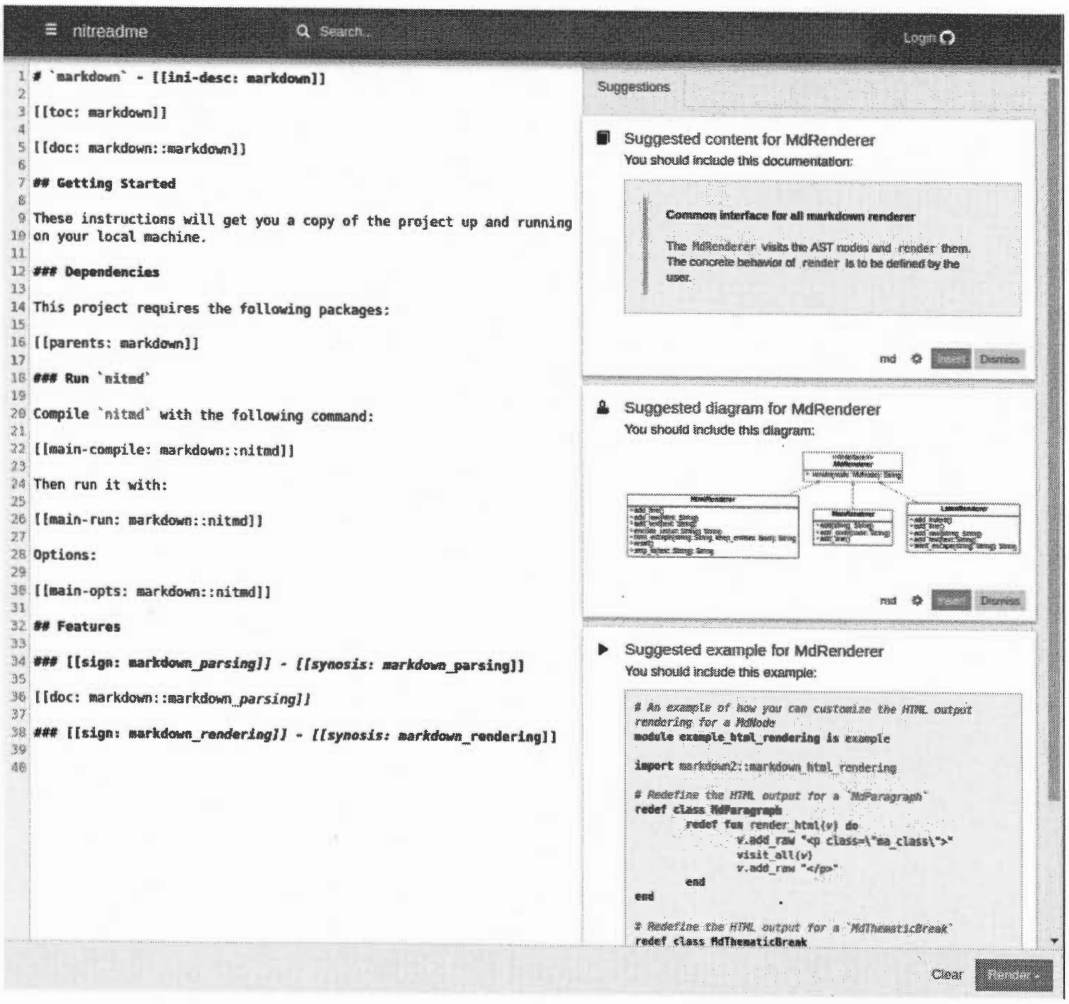


Figure 7.4: Interface utilisateur de l'outil de documentation assistée `nitreadme`.

par des suggestions de formules ou de graphiques à insérer dans la feuille. Lorsque l'utilisateur clique sur une carte de suggestion, une fenêtre s'affiche pour lui permettre de personnaliser son graphique puis de l'insérer dans la feuille de calcul. Un aperçu de l'interface de cet outil est disponible en annexe D.1.

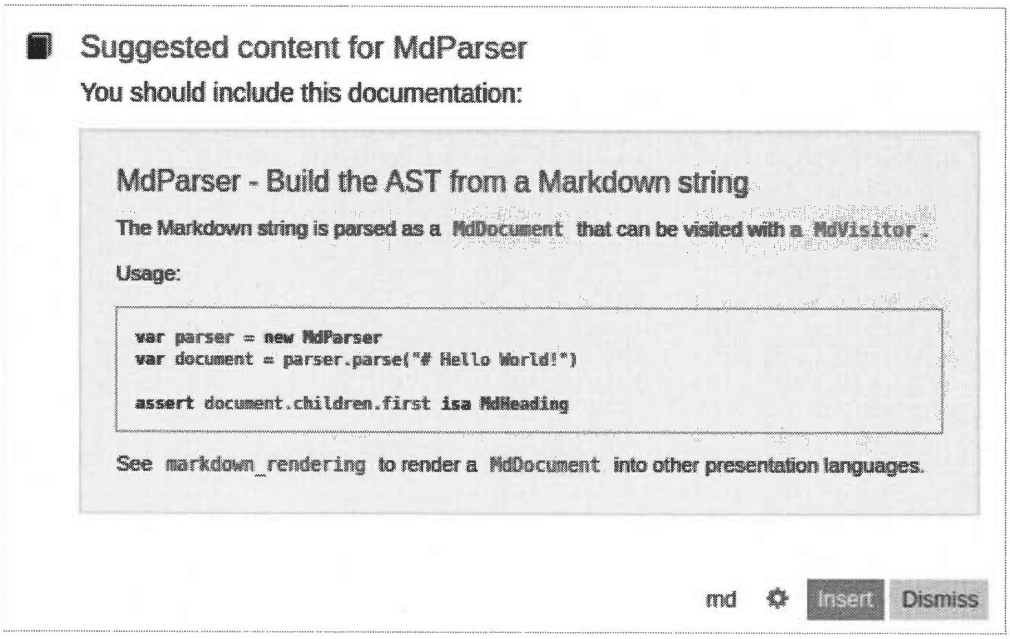
Nous réutilisons ce principe d'affichage par cartes afin de présenter les suggestions liées à la documentation. Comme on peut le voir dans l'interface de notre outil présentée en figure 7.4, la colonne de droite permet à `nitreadme` de présenter les suggestions de documentation que l'écrivain peut sélectionner afin de les appliquer au document README en cours d'écriture.

Pour un même contexte, plusieurs cartes peuvent être affichées simultanément. La colonne dispose d'un ascenseur permettant à l'écrivain de faire défiler les suggestions *triées par ordre de pertinence* par rapport au contenu en cours d'écriture, la carte la plus pertinente selon le contexte se trouvant en haut de la liste.

La figure 7.5 donne un exemple de suggestion de carte relative à l'insertion du commentaire de la classe `MdParser` accompagné d'un lien vers l'entité dans l'auto-documentation d'API servie par `nitweb`. Les actions possibles pour l'écrivain sont présentées en bas à droite de la carte. Le bouton `Insérer` (*Insert*) permet à l'écrivain d'insérer le contenu Markdown de la carte à la position du curseur dans le texte. Le bouton `Ignorer` (*Dismiss*) permet à l'écrivain de rejeter la suggestion dans ce contexte. La carte ne sera alors plus proposée pour la section du document en cours d'écriture.

7.2.2 `nitweb/readit` – Assistant de lecture

Contrairement aux autres modes de l'outil `nitreadme`, le mode assistant de lecture n'est pas destiné aux écrivains mais aux lecteurs. Il s'utilise par l'intermédiaire de l'interface HTML de l'outil `nitweb` sur les pages de visualisation des fichiers README des packages et des groupes Nit (section 6.2).



(a) Carte de suggestion.

```
[[doc: markdown::MdParser]]
```

(b) Contenu Markdown inséré.

Figure 7.5: Exemple de carte d'importation du commentaire de la classe `MdParser`.

Lorsqu'un lecteur survole le contenu d'un fichier README, il peut choisir d'activer le mode assistant de lecture en cliquant sur le bouton *Assistant* (*Assist*). Une fois activé, ce mode affiche une liste déroulante sur la droite du contenu en cours de lecture permettant de suggérer du contenu supplémentaire à lire en fonction de la section survolée par le lecteur.

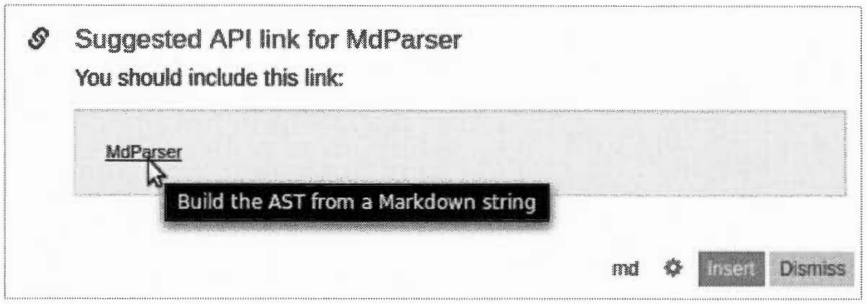
Ces suggestions sont basées sur les cartes de suggestion de contenu descriptif présentées aux écrivains mais sans action de modification possible. Ainsi, lorsqu'un lecteur parcourt un fichier README, l'outil peut lui suggérer des liens vers la documentation d'API, des exemples de code ou encore des diagrammes UML. Une capture d'écran de l'interface de `nitweb/readit` est donnée en annexe, figure C.4.

Cette approche permet d'enrichir la lecture de fichiers README même si ceux-ci n'ont pas été écrits à l'aide de `nitreadme`.

7.3 Suggestions de cartes de documentation

Les insertions de contenu sont faites à l'aide des directives de documentation Nit (section 6.4). Pour chaque suggestion, `nitreadme` produit une directive représentant le contenu à générer lors de la compilation de la documentation vers un format de présentation comme HTML ou LaTeX. Ainsi, lorsque l'écrivain choisit d'ajouter une carte de contenu descriptif dans son document README, l'outil `nitreadme` insère la directive de documentation correspondante à la position du curseur dans le Markdown. Nous générons des directives de documentation plutôt que le contenu définitif afin de faciliter la tenue à jour de la documentation.

Par exemple, lorsque l'écrivain clique sur le bouton `Insérer` (*Insert*) de la carte montrée en figure 7.6a, `nitreadme` introduit la directive de documentation présentée en figure 7.6b à la position du curseur dans le document README en cours d'écriture.



(a) Carte de suggestion.

```
[[markdown::MdParser]]
```

(b) Contenu Markdown inséré.

Figure 7.6: Exemple de carte de suggestion présentant les actions `Insérer` (*Insert*) et `Ignorer` (*Dismiss*) ainsi que le contenu Markdown inséré dans le document README.

Une fois compilée vers HTML, cette directive sera représentée par un lien hypertexte pointant vers la documentation d'API de la classe `markdown::MdParser` dans `nitweb`. L'écrivain peut visualiser le contenu Markdown qui sera inséré en cliquant sur le lien `md` en bas de la carte.

Certaines cartes proposent des options de configuration que l'écrivain peut modifier en fonction de ses besoins. La présence d'options de configuration sur une carte est signalée par le bouton en forme de roue dentée. Lorsque l'écrivain clique sur ce bouton, une fenêtre modale lui propose de modifier les options de la carte. Un exemple de fenêtre modale pour la carte d'insertion de lien vers la documentation d'API est présenté en figure 7.7. Lorsque les options sont modifiées par l'écrivain, la directive de documentation est mise à jour pour représenter ces changements.

Comme l'illustre le diagramme de classes présenté en figure 7.8, les cartes de documentation peuvent se regrouper en trois grandes catégories. Les *cartes d'échafaudage* (*Scaffold*) qui permettent de créer la structure du document selon les

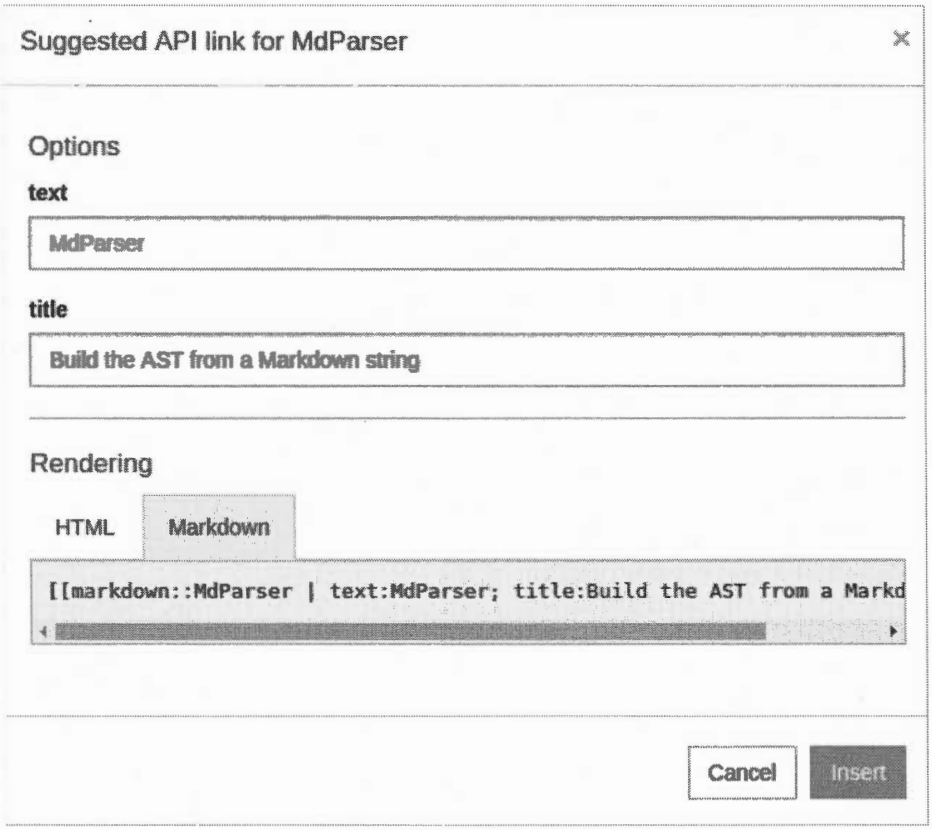


Figure 7.7: Exemple de fenêtre modale de configuration pour la carte de lien.

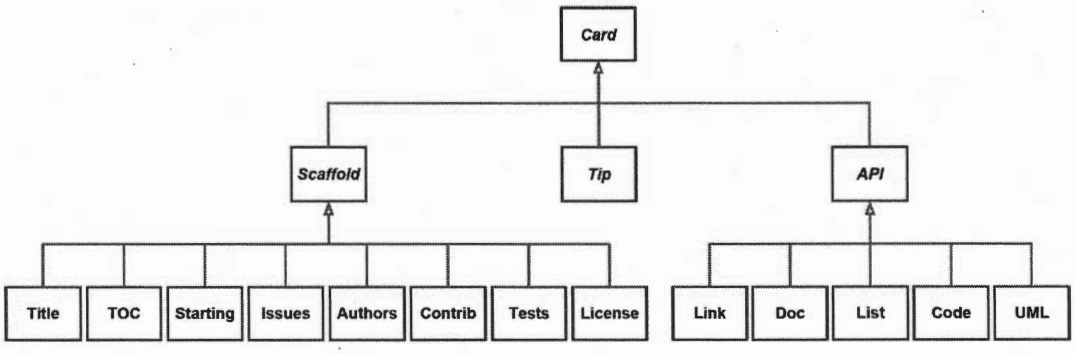


Figure 7.8: Diagramme de classes des cartes de documentation suggérées par `nitreadme`.

sections traditionnelles du README. Les *cartes d'API* (*Scaffold*) qui proposent d'inclure des extraits de l'auto-documentation d'API directement dans le README. Enfin, les *cartes de conseil* (*Tip*), qui sont différentes car elles ne sont pas directement utilisables par l'écrivain. Elle sont générées par `nitreadme` afin de suggérer des changements à appliquer dans le code ou les méta-données pour améliorer la qualité du projet et de sa documentation.

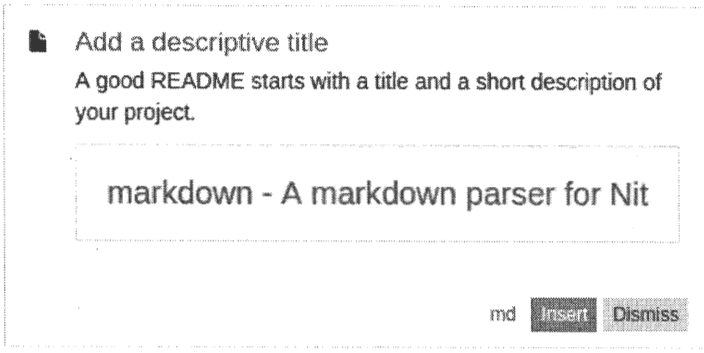
7.3.1 Cartes d'échafaudage

Les *cartes d'échafaudage* (*Scaffold* dans la figure 7.8), permettent d'échafauder — ou construire — la structure générale de la documentation en y insérant les sections principales qui composent un bon README. Le contenu et l'ordre des sections d'échafaudage proposées sont directement tirés de notre étude sur le contenu des fichiers README de GitHub présentée au chapitre 2. `nitreadme` ne suggère des cartes que pour les sections où il est possible de générer et garder synchronisé du contenu à partir du code ou des méta-données.

7.3.1.1 Carte de titre

La *carte de titre* (*Title*) permet à l'écrivain d'insérer automatiquement le nom du projet et sa description succincte. La figure 7.9a présente un exemple pour le projet `markdown` qui contient l'analyseur syntaxique Markdown utilisé par les outils de documentation du langage Nit.

Lorsqu'elle est sélectionnée, cette carte introduit le contenu Markdown présenté en figure 7.9b à la position actuelle du curseur dans le document. Le Markdown inséré contient un titre de niveau 1 avec le lien vers le projet en cours de documentation, ici le projet `markdown`, ainsi que la directive de documentation `ini-desc: markdown` permettant d'importer la description courte du projet depuis la clé `package.desc` dans son fichier de description du package (`package.ini`).



(a) Carte de suggestion.

```
# [[markdown]] - [[ini-desc: markdown]]
```

(b) Contenu Markdown inséré.

Figure 7.9: Exemple de suggestion de titre et de description succincte pour le projet `markdown`.

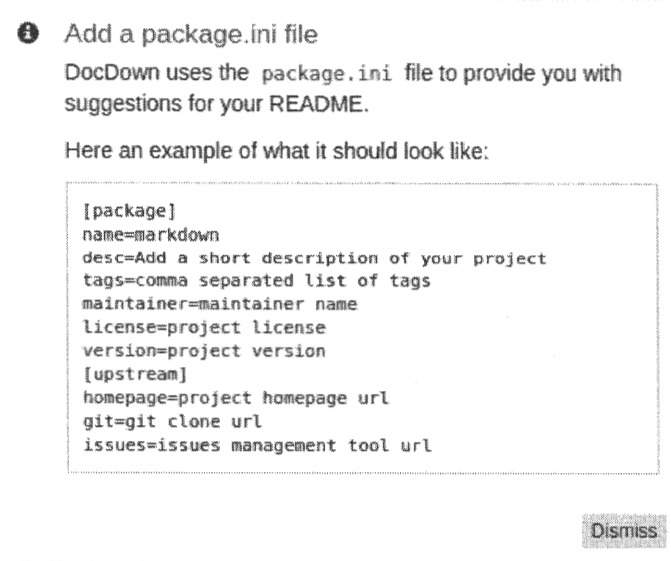


Figure 7.10: Exemple de carte suggérant à l'écrivain d'ajouter le fichier de description du package (`package.ini`) dans son projet.

`nitreadme` peut aussi suggérer des améliorations à son projet afin de fournir les informations de base utilisées pour générer le contenu des cartes. Par exemple, si le projet ne contient pas le fichier `package.ini` qui est utilisé pour extraire certaines informations concernant l'échafaudage du README, alors l'outil affiche une *carte de conseil* (*Tip*) lui suggérant d'ajouter ce fichier tel que présenté en figure 7.10.

Les conseils s'adaptent aux informations déjà fournies par l'écrivain. Par exemple, si le projet contient déjà un fichier `package.ini` mais que celui-ci n'a pas défini la clé `package.desc`, alors l'outil affichera une carte de conseil plus précise (voir annexes E.1 et E.2).

7.3.1.2 Carte de sommaire

La *carte de sommaire* (*TOC*) affiche le sommaire composé des sections de niveau 2 dans le document. Une fois compilée vers un langage de présentation, elle utilise une liste ordonnée de liens pointant vers les sections dans le document.

Un exemple de carte pour le projet `markdown` est présenté en annexe, à la figure E.3a, alors que le contenu Markdown inséré par cette carte est présenté à la figure E.3b.

7.3.1.3 Carte de démarrage

La *carte de démarrage* (*Starting*) présente au lecteur les commandes nécessaires pour débuter avec le projet dont il est en train de lire la documentation. Elle explique les dépendances du projet et comment en récupérer les sources. Dans le cas où le projet contient des exécutables, elle précise comment les compiler et les exécuter.

Un exemple de carte pour le projet `markdown` est donné en figure 7.11, alors que le contenu Markdown inséré par cette carte est donné en figure 7.12.

Figure 7.11: Exemple de carte de démarrage (*Getting Started*) pour le projet `markdown`.

```
## Getting Started

These instructions will get you a copy of the project up
   and running on your local machine.

### Dependencies

This project requires the following packages:
[[parents: markdown]]

### Getting the sources

Clone the source from the git repository:
[[git-clone: markdown]]

### Running [[markdown::nitmd]]

Compile [[markdown::nitmd]] with the following command:
[[main-compile: markdown::nitmd]]

Then run it with:
[[main-run: markdown::nitmd]]

Options:
[[main-opts: markdown::nitmd]]
```

Figure 7.12: Exemple de contenu inséré dans le document pour la carte de démarrage (*Getting Started*) du projet `markdown`.

Le contenu Markdown inséré contient plusieurs directives de documentation afin de présenter les différentes informations que l'utilisateur de la bibliothèque doit connaître pour débuter :

`parents: markdown` liste les dépendances du projet.

`git-clone: markdown` affiche la commande `git` permettant de cloner les sources du projet. L'URL du dépôt est importée depuis la clé `upstream.git` du fichier `package.ini`.

`main-compile: markdown::nitmd` explique comment compiler l'exécutable `nitmd` grâce au compilateur du langage Nit, `nitc`.

`main-run: markdown::nitmd` explique comment lancer l'exécutable `nitmd`. Le synopsis de la commande est importé depuis le fichier MAN de l'exécutable.

`main-opts: markdown::nitmd` liste les options acceptées par l'exécutable `nitmd`. La liste des options et leurs descriptions sont importées depuis le fichier MAN de l'exécutable.

Puisque la carte de démarrage s'appuie sur le contenu du MAN de l'exécutable `nitmd`, un conseil est affiché si le projet ne contient pas le fichier MAN recherché ou si son contenu est incomplet.

7.3.1.4 Carte de problèmes et support

La *carte de problèmes* (*Issues*) insère une section présentant un lien vers l'outil choisi par le développeur, par exemple GitHub. Un exemple d'une telle carte pour le projet `markdown` est donné en annexe (figure E.4a) ; le contenu inséré par la carte (figure E.4b) se base sur la directive de documentation `ini-issues: markdown`. Le lien inséré par cette directive dépend de l'outil utilisé par le développeur et est importé depuis la clé `upstream.issues` du fichier de description du package. Là encore, si le fichier de description du package ne contient pas la clé nécessaire, un conseil est affiché expliquant à l'écrivain comment améliorer son projet.

7.3.1.5 Cartes d'auteurs et contributeurs

`nitreadme` suggère une *carte d'auteurs* (*Authors*) permettant d'importer les listes d'auteurs et de contributeurs depuis le fichier de description du package. Un exemple de carte d'auteurs est donné en annexe (figure E.5). Le nom et le contact de l'auteur principal du projet sont importés depuis la clé `package.maintainer` du fichier de description du package via la directive `ini-maintainer` et la liste des contributeurs depuis la clé `package.more_contributors` via la directive `ini-contributors`. Si l'une de ces deux clés est absente, `nitreadme` affiche un conseil invitant l'écrivain à compléter le fichier `package.ini`.

7.3.1.6 Carte de contribution

`nitreadme` suggère une section contenant un lien vers le dépôt de sources à utiliser et listant les consignes à respecter pour soumettre une nouvelle contribution. Un exemple de *carte de contribution* (*Contrib*) pour le projet `markdown` est donné en annexe, figure E.6.

Le lien vers le dépôt de sources est extrait du fichier de description du package via la clé `upstream.git`. Ce lien est importé grâce à la directive `ini-git: markdown`. Accessoirement, l'écrivain peut fournir un fichier listant les règles de contribution nommé `CONTRIBUTING` qui est importé via la directive `contrib-file: markdown`.

Si la clé `upstream.git` ou le fichier `CONTRIBUTING` sont absents, alors l'outil `nitreadme` affiche un conseil expliquant à l'écrivain comment corriger la situation.

7.3.1.7 Carte de tests

La *carte de tests* (*Tests*) explique à l'utilisateur comment lancer les tests unitaires du projet à l'aide de `nitunit`, l'outil de test du langage Nit. La carte de tests (annexe,

figure E.7) affiche la commande `nitunit` à exécuter pour lancer l'ensemble des tests unitaires pour le projet en cours de documentation. Cette carte s'appuie sur la directive `testing: markdown` se chargeant d'extraire la liste des tests unitaires contenus dans le projet `markdown`.

Si aucun test unitaire n'est trouvé pour le projet, alors `nitreadme` affiche un conseil expliquant à l'écrivain comment en ajouter grâce à l'outil `nitunit`.

7.3.1.8 Carte de licence

La *carte de licence* (*License*) stipule les règles concernant l'inclusion, la modification ou la réutilisation du code dans un autre projet ainsi que les contraintes liées à sa commercialisation. Un exemple pour le projet `markdown` est donné en annexe (figure E.8). Le contenu inséré par la carte est extrait depuis la clé `package.license` du fichier de description du package puis insérée grâce à la directive de documentation `ini-license: markdown`. Accessoirement, l'écrivain peut aussi fournir un fichier `LICENSE` afin d'expliquer les détails de la licence. Ce fichier sera importé par `nitreadme` grâce à la directive `license-file: markdown`.

Si le fichier `LICENSE` est introuvable ou si la clé `package.license` n'est pas définie, alors l'outil `nitreadme` affiche les conseils correspondants.

7.3.2 Cartes de documentation d'API

Les *cartes de documentation d'API* (classe *API* dans la figure 7.8 page 181) offrent des suggestions d'ajout relatives au code source en cours de documentation basées sur ce qui serait généré par des outils d'auto-documentation comme `nitdoc` ou `Javadoc`. Elles correspondent par exemple à des ajouts de liens vers la documentation d'API, à des exemples issus du code source ou encore à des diagrammes UML. Les sous-sections suivantes présentent les différents types de cartes documentation d'API suggérées à l'écrivain par `nitreadme`.

7.3.2.1 Liens vers la documentation d'API servie par `nitweb`

Lorsqu'un écrivain documente son projet, il peut faire référence à des entités présentes dans le code source telles que les modules, les classes et les propriétés. Pour la plupart, ces entités possèdent déjà une documentation détaillée de leur API disponible dans l'outil `nitweb`.

Pour chaque entité détectée dans le texte du fichier README, la *carte de lien* (*Link*) suggère l'insertion d'un lien hypertexte vers sa page de documentation d'API dans `nitweb`. Un exemple de carte suggérant l'inclusion d'un lien vers la documentation d'API de la classe `MdParser` est donné en figure 7.6 (p. 180).

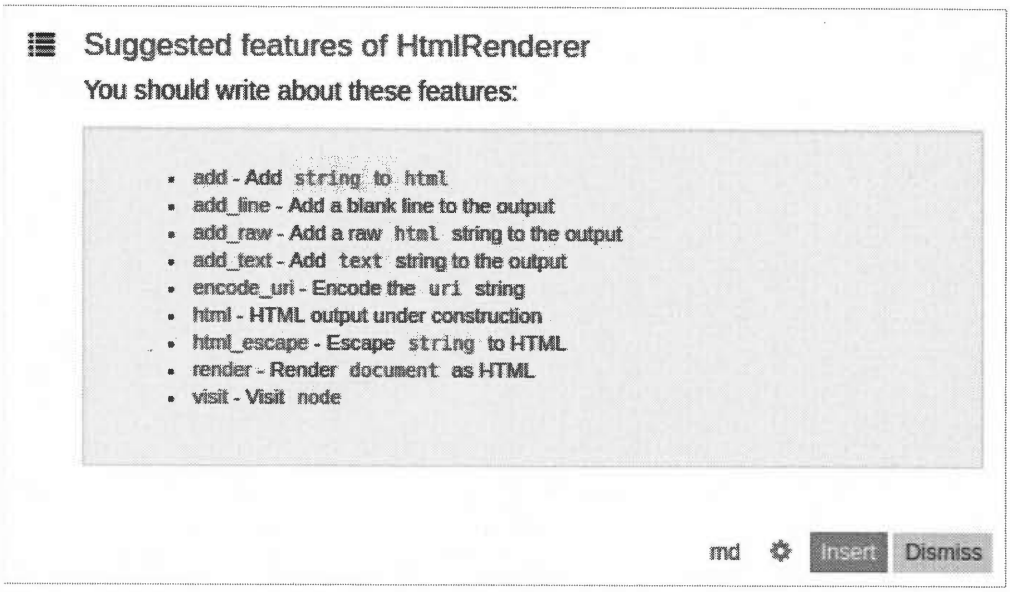
7.3.2.2 Importation de commentaires

La plupart des fonctionnalités sont déjà documentées dans le code source du projet via les commentaires d'en-tête des modules, classes et propriétés. Parfois, ces commentaires peuvent être directement réutilisés dans le README.

Afin d'éviter le copier-coller, et donc limiter les risques de désynchronisation entre la documentation et le code source, les commentaires des entités détectées dans le texte peuvent être directement importés dans le fichier README. C'est l'objectif de la *carte d'importation de commentaires* (*Doc*) telle que présentée en figure 7.5 (p. 178). La directive de documentation sera ensuite remplacée par le contenu du commentaire de l'entité lors de la génération du README dans son format final.

7.3.2.3 Listes d'entités

Afin de documenter les fonctionnalités offertes par un module ou une classe, il est souvent utile de lister les entités les plus importantes qui y figurent. Par exemple, un écrivain voudra lister les classes les plus importantes d'un module, lister tous les attributs d'une classe ou encore juste ses constructeurs.



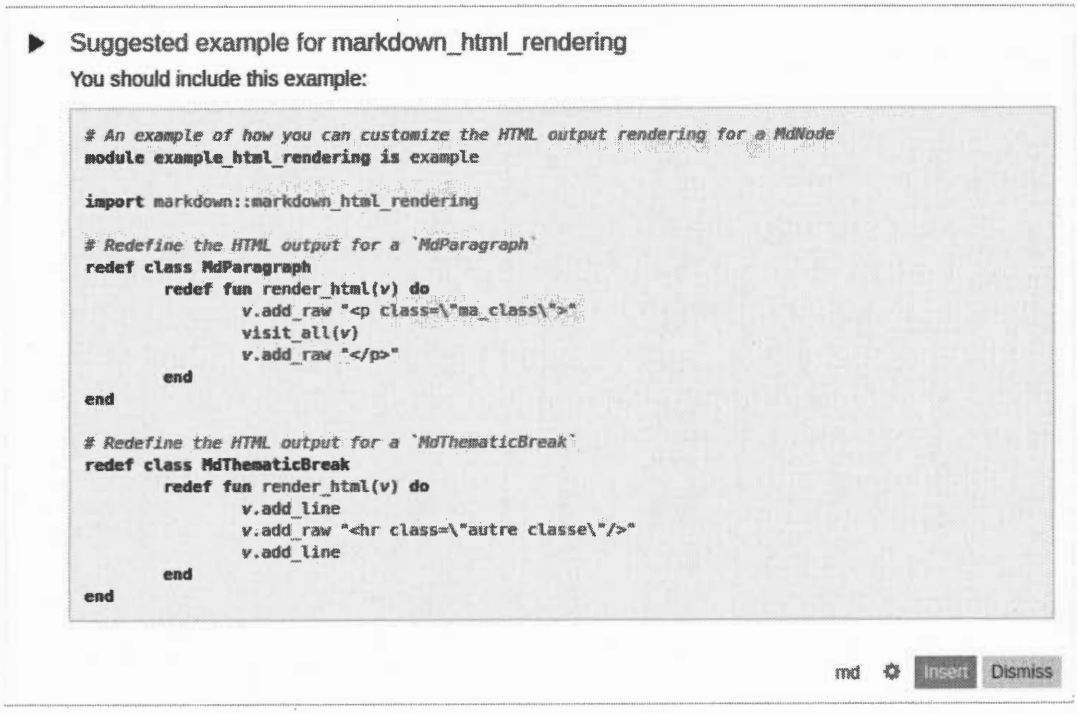
(a) Carte de suggestion.

```
[[defs: markdown::HtmlRenderer]]
```

(b) Contenu Markdown inséré.

Figure 7.13: Exemple de carte de liste de propriétés issues de la classe `HtmlRenderer`.

L'objectif de la *carte de liste d'entités* (*List*) est de composer une telle liste automatiquement depuis le modèle du projet. Le contenu de la liste est extrait depuis les entités trouvées dans le modèle pour un module ou une classe en particulier, et automatiquement mis à jour en cas de changement dans le code source. L'ordre de tri de la liste et les informations affichées sur les entités présentes dans la liste sont configurables. Un exemple de cette carte est donné en figure 7.13.



(a) Carte de suggestion.

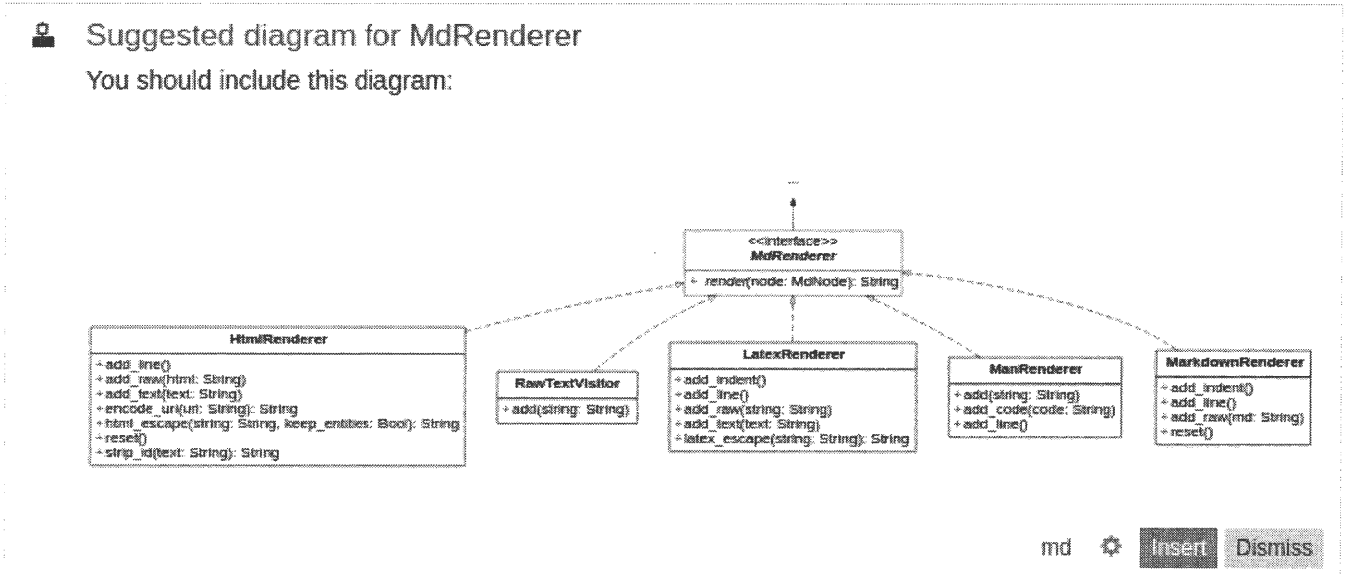
```
[[code: markdown::example_html_rendering]]
```

(b) Contenu Markdown inséré.

Figure 7.14: Exemple de carte d'exemple pour la classe `HtmlRenderer`.

7.3.2.4 Importation de code et d'exemples

Le code peut être importé directement dans le fichier README à l'aide de la *carte de code* (*Code*). Cette carte est particulièrement utile pour inclure le code source des exemples annotés par `is example` dans la documentation. La figure 7.14a insère un exemple d'utilisation de la classe `HtmlRenderer`. Une fois sélectionnée par l'écrivain, cette carte ajoute la directive de documentation présentée dans la figure 7.14b dans le document README.



(a) Carte de suggestion.

```
[[uml: markdown::MdRenderer]]
```

(b) Contenu Markdown inséré.

Figure 7.15: Exemple de carte de diagramme de classes UML pour la classe `markdown::MdRenderer`.

7.3.2.5 Diagrammes UML

Afin d'encourager l'utilisation des diagrammes de classes ou de packages dans la documentation, `nitreadme` suggère à l'écrivain des diagrammes UML prêts à l'emploi en fonction des modules et classes décrits dans le fichier README grâce à la *carte de diagramme UML* (*UML*).

La figure 7.15a donne un exemple de diagramme de classes présentant les ancêtres et descendants de la classe `markdown::MdRenderer`. Une fois sélectionnée par l'écrivain, cette carte insère la directive de documentation présentée à la figure 7.15b qui permet de maintenir le diagramme à jour automatiquement.

7.4 Conclusion

Dans ce chapitre, nous avons présenté, sous le nom de `nitreadme`, notre solution pour la création et la maintenance de fichiers README, Cet outil peut être utilisé par l'intermédiaire de la ligne de commande ou d'un navigateur web pour créer, améliorer et maintenir les fichiers README grâce à la suggestion de cartes de documentation.

Les suggestions proposent du contenu à l'écrivain qu'il peut insérer dans son fichier README afin d'en améliorer la structure et la documentation des entités de l'API. Nous avons listé ces suggestions de cartes et décrit leur contenu ainsi que les directives de documentation utilisées pour les créer et les maintenir à jour.

Le tableau 7.1 compare notre solution avec celles présentées dans notre état de l'art des générateurs de fichiers README présenté au chapitre 3. Comme on peut le constater, `nitreadme` offre des fonctionnalités dans toutes les dimensions. L'outil suggère des cartes de documentation relatives à l'échafaudage du fichier, la documentation de son API, l'inclusion d'exemples et d'abstractions sous la

Tableau 7.1: Comparaison de `nitreadme` avec les outils de génération de fichiers README.

Légende: La comparaison est établie selon le support de l'outil pour l'échafaudage du contenu (**Échaf.**), la présentation des éléments de l'API (**API**), la présentation d'exemples (**Exs.**) et l'utilisation d'abstractions (**Abs.**)

	Échaf.		API		Exs.		Abs.	
Outils	Suggestion	Synchronisation	Suggestion	Synchronisation	Suggestion	Synchronisation	Suggestion	Synchronisation
nitreadme	•	•	•	•	•	•	•	•
Go ReadMe	•							
Readme Generator	•							
generate-readme	•	•						
grunt-readme-generator	•	•						
README.md Generator	•	•						
Rebecca				•		•		
verb				•				•

forme de diagrammes UML. L'utilisation des directives de documentation permet d'assurer la synchronisation du fichier README avec le fichier source.

Dans les prochains chapitres, nous expliquerons les rouages internes de notre solution, laquelle repose sur un alignement entre le contenu du fichier README et les entités issues de l'API (chap. 8) pour supporter un processus de sélection des cartes de suggestions (chap. 9).

CHAPITRE VIII

`DOC_ALIGN` : UNE BIBLIOTHÈQUE POUR ALIGNER LE CONTENU DU README AVEC CELUI DE L'API

Notre approche de génération des suggestions de documentation par `nitreadme` (chap. 7) repose sur l'établissement de correspondances — d'un *alignement* — entre le texte trouvé dans le fichier README et les entités du code Nit. Ce chapitre présente l'implémentation de la bibliothèque `doc_align` conçue pour effectuer cet alignement. Cette bibliothèque est importée par l'outil `nitreadme`. La figure 8.1 montre sa position dans l'ensemble de l'écosystème Nit.

La section 8.1 présente tout d'abord une revue de la littérature sur l'alignement de la documentation avec le code source.

La section 8.2 présente le corpus des fichiers README extraits du projet Nit que nous avons utilisé pour valider les différentes étapes de notre processus ainsi que son résultat final.

Le processus d'alignement du contenu des fichiers README avec les entités de l'API Nit, illustré à la figure 8.2, se déroule en deux temps : i) *l'étape hors ligne* où le contenu des fichiers Nit passés à `nitreadme` est indexé au lancement de l'outil (boites grises) ; ii) *l'étape en ligne* où le document README en cours d'écriture est analysé et dont le contenu est aligné avec l'index produit préalablement (boites bleues). La suite de ce chapitre présente chacune de ces étapes dans l'ordre où elles sont exécutées.

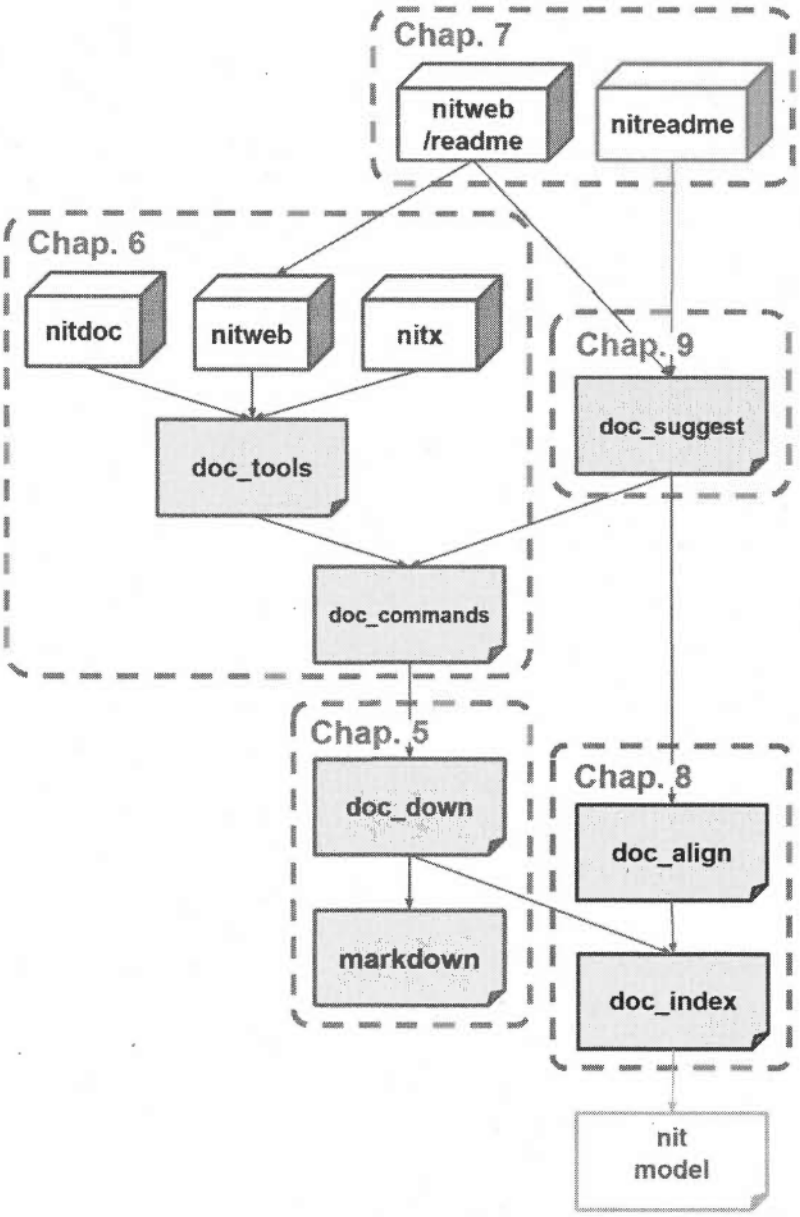


Figure 8.1: Place des modules `doc_index` et `doc_align` dans l'écosystème de documentation Nit.

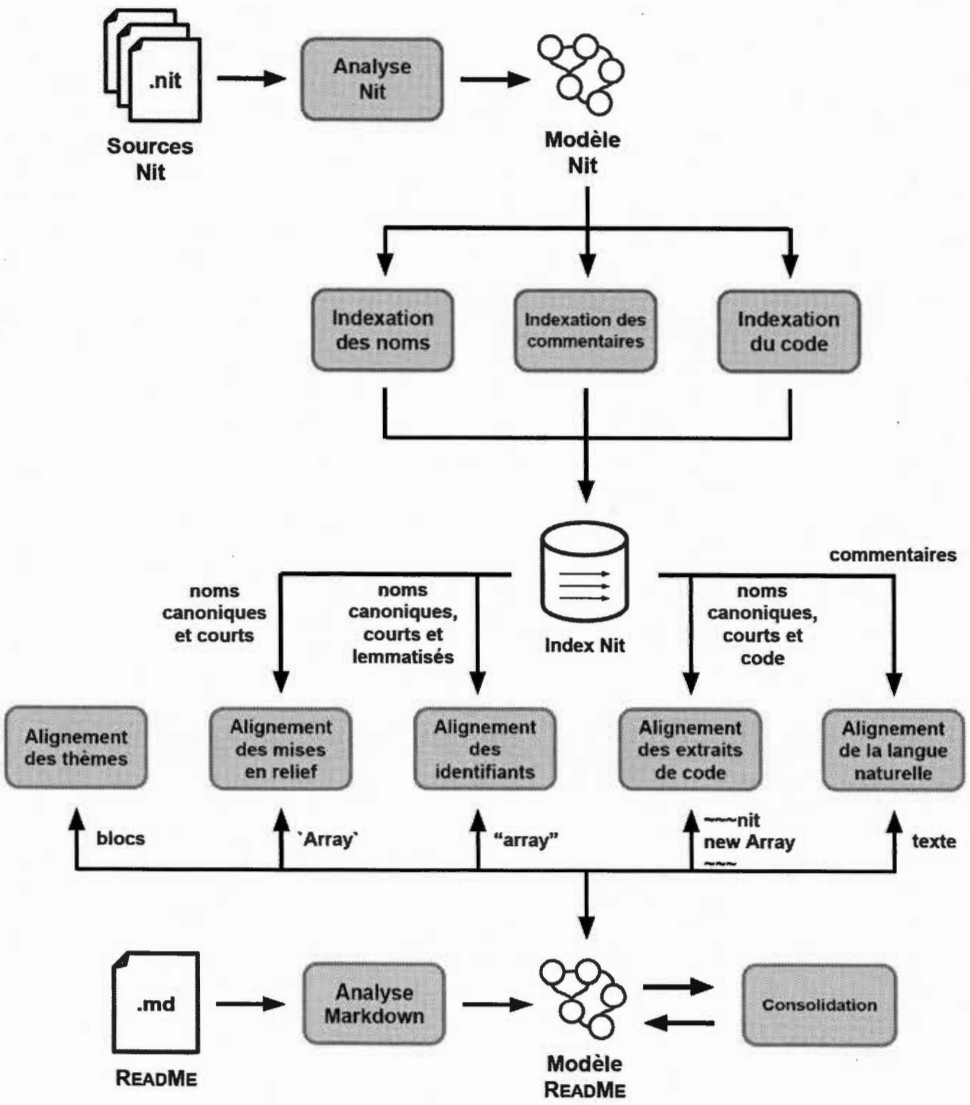


Figure 8.2: Vue générale du processus d'alignement du contenu des fichiers README avec les entités de l'API Nit. Les boites grisées représentent les étapes réalisées *hors ligne*. Les boites bleues sont les étapes réalisées au cours de la rédaction par l'écrivain.

L'**analyse Nit** (section 8.3) compose le *modèle Nit*, une représentation du contenu du code, c'est-à-dire des entités de l'API à aligner avec le fichier README.

Vient ensuite la phase d'**indexation du modèle Nit** (section 8.4) qui se décompose elle-même en trois sous-phases :

8.4.1 : L'**indexation des noms**, pour retrouver une entité Nit à partir de son nom ;

8.4.2 : L'**indexation des commentaires**, pour retrouver une entité Nit à partir d'un vecteur représentant son commentaire ;

8.4.3 : L'**indexation du code**, pour retrouver une entité Nit, un exemple ou un test unitaire à partir d'un vecteur représentant son implémentation ;

L'**analyse Markdown** (section 8.5) construit le modèle README représentant le contenu du fichier à aligner avec le modèle Nit. Puis s'exécutent les différentes sous-phases du **processus d'alignement** (section 8.6) qui se font lors de l'*étape en ligne* :

8.6.1 : L'**alignement des thèmes** utilise la notion de *thème*, présentée au chapitre 2, pour abstraire le rôle des sections et blocs Markdown du fichier README ;

8.6.2 : L'**alignement des mises en relief de code** extraites du Markdown avec les entités de l'index Nit grâce aux noms canoniques et courts ;

8.6.3 : L'**alignement des identifiants** présents dans le texte Markdown avec les entités de l'index Nit grâce aux noms canoniques, courts et lemmatisés ;

8.6.4 : L'**alignement des extraits de code** utilise l'index Nit pour aligner les extraits de code du Markdown avec les entités de l'API ;

8.6.5 : L'**alignement de la langue naturelle** dans le texte Markdown avec les entités Nit grâce à l'indexation des commentaires ;

8.6.6 : La **consolidation** combine le résultat des phases précédentes pour propager l'alignement suivant la structure du document Markdown.

Enfin, la section 8.7 présente les menaces liées à la validité de notre étude.

8.1 État de l'art de l'alignement entre le code source et la documentation

Établir le lien entre les artefacts de documentation et le code source n'est pas une problématique nouvelle. Depuis le début des années 2000, de nombreuses approches ont été développées utilisant des techniques différentes et pour des objectifs différents. Dans cette section, nous résumons l'état de l'art de ce domaine.

8.1.1 Alignement de la langue naturelle

L'alignement de la langue naturelle consiste à établir le lien (les correspondances) entre la langue naturelle trouvée dans les artefacts de documentation et le contenu des APIs.

Antoniol *et al.* (2002) tentent d'établir le lien entre le code source de bibliothèques C++ et les concepts de plus haut niveau exprimés dans la documentation. Leur approche se base sur un modèle probabiliste pour établir les correspondances entre les termes utilisés dans le code et ceux utilisés dans la documentation.

Marcus et Maletic (2003) utilisent l'indexation sémantique latente (*latent semantic indexing*) pour établir ce même lien entre le vocabulaire trouvé dans le manuel d'une bibliothèque C++ et les identifiants du code source.

Ces approches, bien que pouvant s'adapter à tout type de vocabulaire, présentent un inconvénient majeur : la précision des résultats. En effet, l'alignement de la langue naturelle entraîne un grand nombre de faux positifs, ce qui limite son utilité pour plusieurs problèmes (Bacchelli *et al.*, 2010).

Dans un article de 2014, Cleland-Huang *et al.* estimaient que l'utilisation des approches traditionnelles d'analyse de la langue naturelle avait atteint un plateau

en terme de précision et que les futurs travaux de la communauté scientifique devraient s'orienter dans une autre direction (Cleland-Huang *et al.*, 2014). Face à ce constat, d'autres études ont tenté d'améliorer la précision de l'alignement par des techniques diverses. Par exemple, Tsuchiya *et al.* (2015) utilisent une rétroaction de la part de l'utilisateur (*user feedback*) pour écarter les résultats non-pertinents, alors que Vinárek *et al.* (2014) mettent à profit des modèles de domaines (*domain models*). Plus récemment, la démocratisation de l'intelligence artificielle a permis de nouvelles avancées dans ce domaine. Ainsi, Ye *et al.* (2016), Uddin et Robillard (2017) et Cao *et al.* (2018) proposent l'utilisation de l'apprentissage machine (*machine learning*) pour filtrer automatiquement les faux positifs.

8.1.2 Alignement des identifiants

Bacchelli *et al.* (2010) montrent qu'en fonction du type de document, l'utilisation des identifiants seuls produit de meilleurs résultats que les approches plus complexes. Ils utilisent des expressions régulières pour extraire les identifiants depuis les listes de courriels de développeurs (*developers mailing lists*) et trouver les mots qui ont une correspondance exacte avec les entités du code source. Leurs résultats indiquent que dans le contexte des courriels, les entités du code source sont plus régulièrement mentionnées par leur nom exact que par des synonymes.

Dagenais et Robillard (2012) utilisent eux aussi des expressions régulières pour extraire les identifiants à la fois des listes de courriels et de la documentation pour des projets Java. Ils montrent que l'utilisation d'expressions régulières seules est insuffisante pour résoudre les ambiguïtés quand un même identifiant peut correspondre à plusieurs entités du code. Ils utilisent le contexte du document et sa structure pour filtrer les résultats et augmenter ainsi la précision de leur approche.

Rigby et Robillard (2013) font le même constat pour les messages issus de Stack-Overflow. Un grand nombre d'identifiants peut être trouvé dans le contenu de

la documentation informelle mais ces identifiants peuvent correspondre à un grand nombre d'entités dans le code. Le problème est alors de déterminer lesquels exactement. Ils réutilisent la même approche que Dagenais et Robillard (2012) pour extraire les identifiants et les filtrer en fonction du contexte du document et de sa structure.

Ces résultats montrent que le vocabulaire utilisé dans les artefacts de documentation peut être aligné directement avec le contenu de l'API. Le problème se situe plutôt au niveau de la sélection de la bonne entité quand le nom engendre une ambiguïté.

8.1.3 Alignement des extraits de code

Outre la langue naturelle et les identifiants, les artefacts de documentation contiennent aussi des extraits de code (*code snippets*). Ces extraits de code représentent une autre source d'information pour aligner le contenu de la documentation avec celui de l'API.

Les extraits de code sont présents aussi bien dans la documentation officielle que dans la documentation informelle. Deursen et Kuipers (1999) les utilisent pour l'alignement de la documentation officielle de projets COBOL, Bacchelli *et al.* (2010) et Dagenais et Robillard (2012) le font dans les chaînes de courriels. Rigby et Robillard (2013) les utilisent pour les fils de discussion StackOverflow. Subramanian *et al.* (2014) font de même pour les échanges sur GitHub (*issues discussions*).

Contrairement à la langue naturelle, les extraits de code apportent un contexte supplémentaire à l'utilisation des identifiants. En effet, les identifiants qu'ils contiennent sont utilisés dans un cadre précis tel une directive d'importation, une création d'instance ou un appel de méthode.

8.1.4 Le cas des fichiers README Nit

L'efficacité d'une approche dépend avant tout du type de document analysé, de son contenu et de l'objectif recherché par l'alignement. Plusieurs des approches présentées plus haut sont conçues pour fonctionner sous l'hypothèse *d'un monde ouvert*. L'alignement des messages issus de StackOverflow ou d'autres plate-formes de questions/réponses implique une base de code très large. En effet, il n'est pas rare que les utilisateurs suggèrent des solutions basées sur plusieurs bibliothèques.

Dans le cas des fichiers README, le contexte est bien défini et généralement plus restreint : il s'agit du package en cours de documentation par l'écrivain. Ce *monde clos* permet donc de limiter les ambiguïtés causées par l'alignement des noms.

La langage Nit, notre cas d'étude, permet aussi d'éviter quelques écueils que rencontrent les autres approches. Dans sa spécification, Nit stipule qu'un package ne peut contenir deux groupes, deux modules ou deux classes portant le même nom. Contrairement aux langages Java, JavaScript et C++, Nit permet donc d'éliminer *de facto* certaines ambiguïtés qui pourraient survenir dans un autre langage.

La spécification de la documentation d'API du langage Nit (chap. 5) permet d'éviter d'autres écueils :

— L'utilisation des mises en relief de code est encouragée pour référencer les entités de l'API, soit par leur nom canonique, soit par leur nom court. La présence de ces mises en relief permet donc d'isoler plus facilement les mots du texte qui doivent être liés au code source.

— Les extraits de code Nit présentés dans la documentation ont plus de chance d'être compilables grâce à l'utilisation de l'outil de tests `nitunit` (section 5.3). Il est donc plus facile d'en examiner le contenu et de le lier aux entités de l'API.

En contrepartie, les fichiers README contiennent aussi des informations qui sont difficilement alignables avec la base de code. Comme nous l'avons montré dans le chapitre 2, certaines sections du README font référence à l'explication du domaine, aux procédures d'installation, à la gestion des problèmes ou encore à la licence. Notre alignement doit aussi prendre en compte ces dimensions.

Enfin, notre objectif est bien particulier. Nous souhaitons suggérer des cartes de documentation selon toutes les dimensions du fichier README, pas seulement sur la présentation du code.

Les approches existantes d'alignement se contentent de trouver quels sont les éléments du code qui sont cités dans le texte et cherchent à obtenir la précision maximale pour ces éléments. Dans notre cas, nous cherchons un maximum de liens sur les blocs du fichier Markdown en fonction du code mais aussi des autres sujets tels que l'installation, la présentation d'exemples, etc.

Face à ces différences, il nous faut trouver quelle combinaison d'informations est exploitable en fonction du contenu du fichier README et de l'alignement que nous souhaitons obtenir.

8.2 Corpus de fichiers README du langage Nit

Pour pouvoir tester et comparer les approches d'alignement du Markdown avec le contenu de l'API, il nous faut un ensemble de fichiers README à analyser. Nous avons choisi ceux des packages du langage Nit comme sujet d'étude. En effet, le langage Nit possède déjà des fichiers README permettant de documenter les packages se trouvant dans le dépôt principal du langage [1].

Nous avons constitué ainsi un corpus composé de 18 packages Nit (bibliothèques et programmes) munis d'un fichier README. Le tableau 8.1 présente chacun de ces packages et indique la taille de chaque fichier README — en nombre de lignes de documentation (LOD).

---

1. `http://github.com/nitlang/nit`

Tableau 8.1: Description des packages composant le corpus des fichiers README de packages Nit.

Légende: **LOD** (*Lines Of Documentation*) : nombre de lignes dans le fichier README du package.

Package	LOD	Description
`actors`	71	Modèle Acteurs pour la programmation concurrente en Nit
`ai`	15	Bibliothèque d'algorithmes utilisés en intelligence artificielle
`android`	165	Cadre de développement d'applications Android en Nit
`app`	209	Cadre de développement d'applications mobiles en Nit
`gamnit`	64	Cadre de développement de jeux et applications multi-médias
`geometry`	57	Structures géométriques
`github`	72	Connecteur à l'API JSON/REST de la plateforme GitHub
`ios`	34	Cadre de développement d'applications OSX en Nit
`json`	240	Bibliothèque JSON pour Nit
`markdown`	34	Compilateur Markdown pour Nit
`nitcorn`	91	Cadre de développement d'applications Web en Nit
`nlp`	73	Connecteur à l'API JSON/REST de Stanford CoreNLP
`popcorn`	856	Cadre de développement d'applications Web en Nit
`posix`	6	Services compatibles POSIX
`pthreads`	33	Support des fils d'exécution POSIX
`sdl2`	14	Interface native à la bibliothèque graphique SDL 2.0
`serialization`	310	Services de sérialisation et désérialisation pour les objets Nit
`vsm`	105	Bibliothèque de modélisation vectorielle

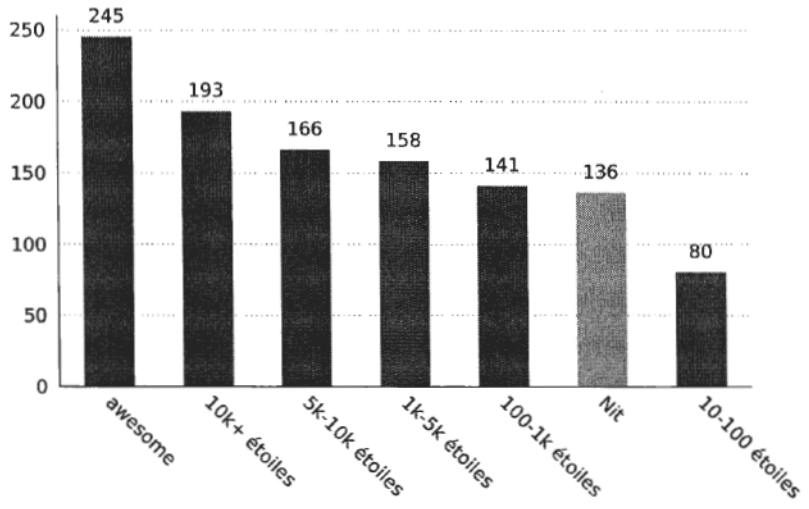


Figure 8.3: Comparaison du nombre de lignes des fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2.

Le projet Nit dans son ensemble, contenant les bibliothèques étudiées, compte 178 étoiles sur GitHub. En l'absence d'étoiles sur chaque bibliothèque, nous avons comparé la taille moyenne des fichiers README Nit avec celle des fichiers issus de GitHub (présentés au chapitre 2) dans la figure 8.3. Les fichiers README du corpus Nit contiennent en moyenne 136 lignes, soit 50 lignes de plus que les fichiers README des projets comptant entre 10 et 99 étoiles et quelques lignes de moins que les projets comptant entre 100 et 999 étoiles.

8.2.1 Structure des fichiers README du corpus

La figure 8.4 donne le nombre moyen d'en-têtes de sections dans les fichiers README du corpus Nit en comparaison avec les fichiers issus de GitHub (chap. 2). Les projets Nit contiennent en moyenne 6.8 en-têtes Markdown, soit le même nombre que les projets comptant entre 10 et 99 étoiles (moyenne : 6.7).

Le niveau des en-têtes est présenté en figure 8.5. Tous les fichiers README du corpus Nit comportent exactement un en-tête de niveau 1 correspondant au titre

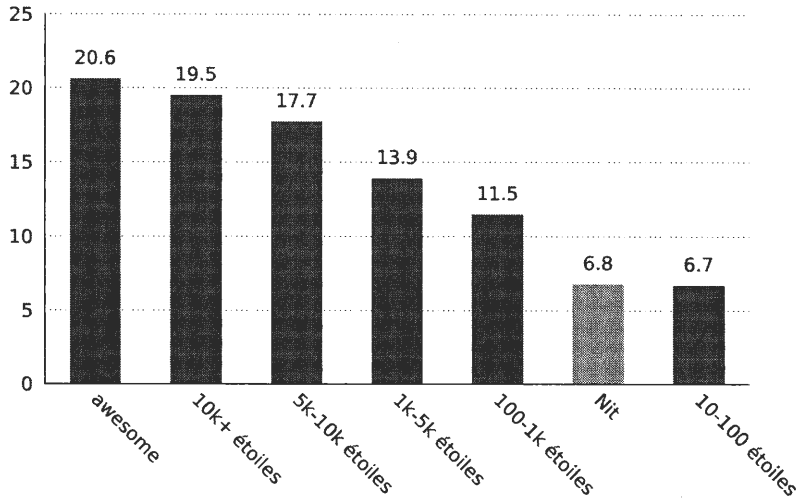


Figure 8.4: Comparaison du nombre d'en-têtes de sections dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2.

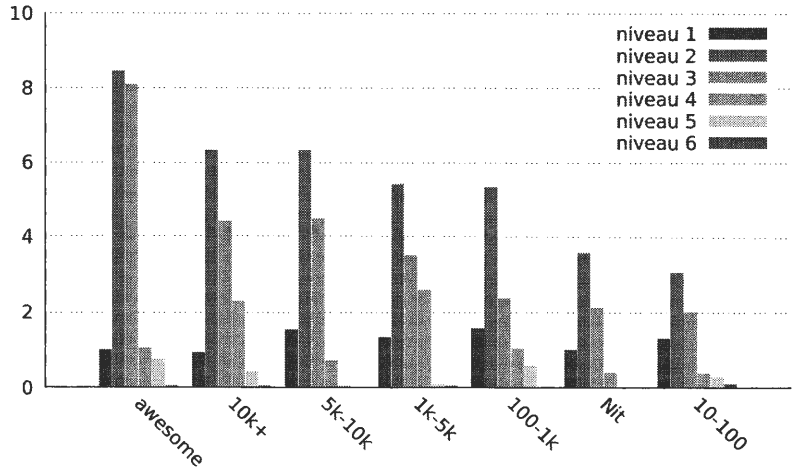


Figure 8.5: Comparaison du niveaux des en-têtes de sections dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2.

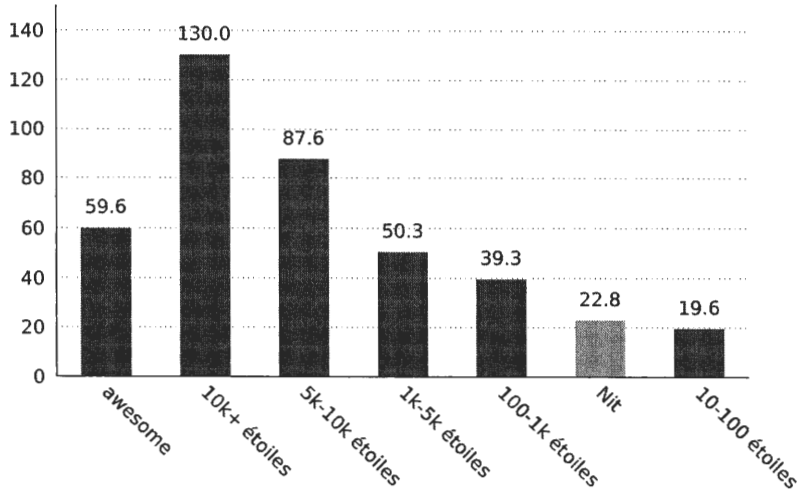


Figure 8.6: Comparaison du nombre de paragraphes dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2.

du document. Ils comptent en moyenne 3.6 en-têtes de niveau 2, 1.8 en-têtes de niveau 3 et 0.4 en-têtes de niveau 4. Il n'y a aucun en-tête de niveau 5 ou 6. Ces moyennes sont comparables avec celles des fichiers README des projets issus de GitHub comptant entre 10 et 99 étoiles.

Enfin, la figure 8.6 indique le nombre moyen de paragraphes dans les fichiers README du corpus Nit en comparaison avec ceux des projets issus de GitHub. Les fichiers README du corpus Nit contiennent en moyenne 22.8 paragraphes (412 occ.), soit un peu plus que les fichiers README des projets comptant entre 10 et 99 étoiles.

8.2.2 Utilisation des extraits de code

Comme nous l'avons montré dans le chapitre 2, les extraits de code Markdown permettent d'inclure des exemples d'utilisation. La figure 8.7 compare la présence des extraits de code dans les fichiers README du corpus Nit et dans ceux du corpus GitHub.

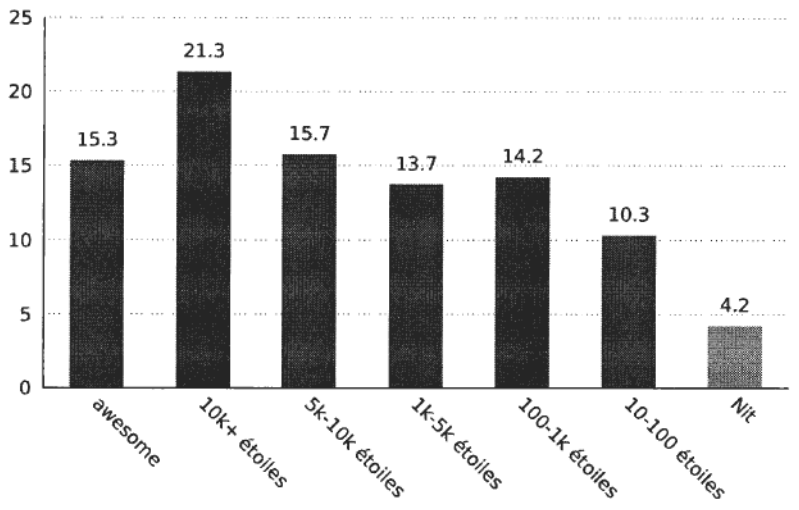


Figure 8.7: Comparaison du nombre d'extraits de code dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2.

En moyenne, le corpus Nit comprend 4.2 extraits de code par README soit presque deux fois moins que les projets comportant de 10 à 99 étoiles (moyenne de 10.3). Tous les extraits de code utilisés dans les fichiers README Nit sont identifiés par des barrières plutôt que par l'indentation — les barrières permettent à l'écrivain d'identifier le langage utilisé dans les extraits de code.

Notre corpus contient un total de 75 extraits de code Markdown dont 60 (80.0% des extraits de code) correspondent à du Nit. Parmi ces 60 extraits, 48 sont effectivement compilables grâce au compilateur `nitc`, ce qui représente 80.0% des extraits de code Nit. Treize (13) utilisations (17.3% des utilisations) correspondent à des extraits de code Shell utilisés pour présenter les procédures d'installations, de compilation et d'exécution. Enfin, deux (2) extraits contiennent autre chose que du code.

8.2.3 Utilisation de la mise en relief de code

Comme nous l'avons observé dans le chapitre 2, les mises en relief de code sont généralement utilisées dans les fichiers README pour faire référence à des entités

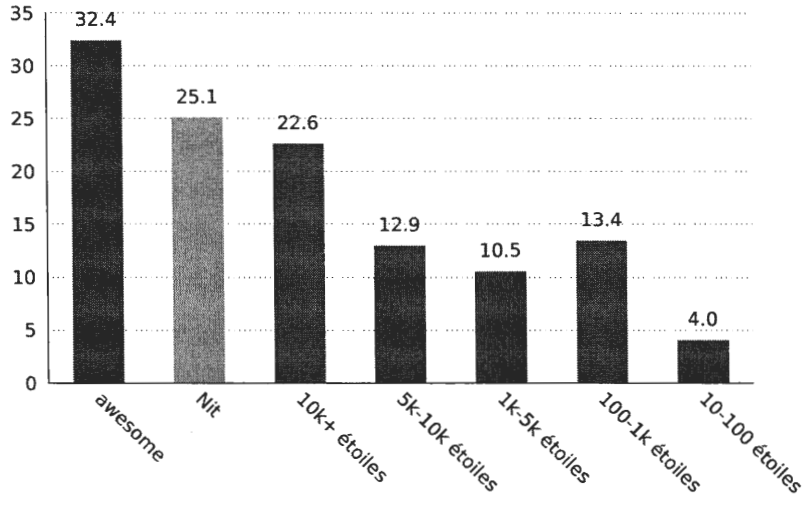


Figure 8.8: Comparaison du nombre de mises en relief de code dans les fichiers README Nit avec ceux issus de GitHub analysés dans le chapitre 2.

de l'API, soit par leur nom qualifié, soit par leur nom court.

La figure 8.8 compare l'utilisation de la mise en relief de code dans le corpus Nit avec l'utilisation faite dans les projets du corpus GitHub. En moyenne, les fichiers README du corpus Nit contiennent 25.1 mises en relief de code, soit un peu plus que les projets GitHub comptant plus de 10 000 étoiles (moyenne de 22.6) mais moins que les projets issus de la liste *Awesome* (moyenne de 32.4).

On dénombre un total de 422 occurrences de mise en relief de code dans notre corpus. Son utilisation la plus fréquente correspond à la présentation de noms d'entités issues du code source. On compte 324 occurrences (71.7% des utilisations) de ce type regroupées en trois catégories :

— **Noms** : 266 noms courts — par ex., `Array` ;

— **Noms Qualifiés** : 35 noms qualifiés — par ex., `core::Array` ;

— **Signatures** : 23 signatures — par ex., `Array[Int]` ou `Array::add(e)`.

La présentation de chemins et de noms de fichiers représente une autre partie importante de l'utilisation des mises en relief de code avec 53 occurrences (11.7% des

utilisations).

On trouve aussi des références vers des annotations du code (32 occ.) comme `test`, `serialize` ou `threaded` et quelques noms de commandes liées aux outils Nit comme `nitc`, `nitmd` ou `nitserial` (11 occ.).

8.2.4 Conclusion sur les caractéristiques du corpus

Les blocs Markdown les plus utilisés dans le corpus Nit sont les paragraphes (412 occ.), les en-têtes de sections (123 occ.) et les extraits de code (75 occ.).

Les fichiers README du corpus Nit contiennent une moyenne de 2.4 listes (ordonnées ou non) pour un total de 44 occurrences, ce qui est bien en dessous de la moyenne de 8.2 listes par fichier README pour les projets comptant entre 10 et 99 étoiles. Aucun bloc de citation et aucune règle n'apparaît dans le corpus Nit.

Concernant les constructions en ligne, ce sont les mises en relief de code qui sont les plus fréquentes avec 422 occurrences. Les autres constructions en ligne du langage Markdown sont rarement utilisées. Notre corpus ne contient que 38 occurrences de la mise en relief de texte, deux occurrences de liens et une seule occurrence d'image.

La constitution et l'utilisation de la syntaxe Markdown des fichiers README du corpus Nit permet de les comparer à ceux des projets issus de GitHub comptant entre 10 et 99 étoiles. Nous pensons que ces projets sont assez représentatifs de petits projets nécessitant le support d'un outil d'amélioration des fichiers README et composent donc un corpus idéal pour nos expérimentations.

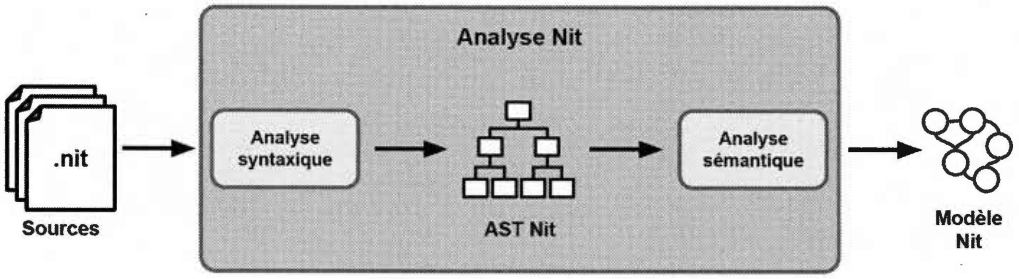


Figure 8.9: Processus d'analyse des sources Nit pour la création du modèle.

8.3 Analyse du code de l'API et création du modèle Nit

La figure 8.9 présente le processus d'analyse du code Nit. Tout d'abord, le code source Nit passé à l'outil `nitreadme` est analysé à l'aide de l'analyseur syntaxique du langage Nit pour créer un arbre de syntaxe abstrait (ou AST, *Abstract Syntax Tree*). Cet arbre est ensuite vérifié par l'analyseur sémantique du langage afin d'obtenir le modèle Nit. Dans cette phase, nous réutilisons les analyseurs syntaxique et sémantique de la chaîne de compilation Nit.

Le modèle Nit est une abstraction des informations trouvées dans le code source représentées sous la forme d'un graphe : les entités de l'API Nit telles que les packages, les modules, les classes et les propriétés représentent les nœuds du graphe ; les relations entre les entités comme l'imbrication, l'importation de modules ou l'héritage de classes représentent les arêtes.

Cette abstraction permet aux outils de la chaîne de compilation d'effectuer des analyses de plus haut niveau que le code source. Par exemple, de vérifier qu'un nom de classe ou de propriété existe bel et bien ou qu'une méthode est accessible par une classe étant donnée ses relations d'héritage.

Le modèle propose en outre des liens vers les différents éléments nécessaires à l'analyse du code et de la documentation, par exemple :

— Le contenu Markdown des commentaires des entités (section 5.2) ;
— Le contenu des fichiers de description des packages (section 5.1.2) ;
— Les nœuds de l'AST liés à chaque entité du code.

Ce modèle, tel qu'il est construit par l'analyseur sémantique du langage Nit, est réutilisé afin de construire l'index nécessaire à l'alignement du contenu des fichiers README avec les entités issues du code.

8.4 Indexation du modèle Nit avec `doc_index`

L'indexation du code de l'API se réalise *hors ligne*. Cette étape peut être relativement longue — en fonction du nombre et de la taille des fichiers Nit à traiter — et se fait en amont du travail de rédaction par l'écrivain. Comme l'alignement se fait *en ligne*, pendant la rédaction, il faut que le processus d'alignement soit rapide afin que l'utilisation de l'outil soit la plus fluide possible. L'analyse du code *hors ligne* permet donc la création d'un index qui sera utilisé pour accélérer l'alignement *en ligne*.

Pour être indexées, les entités du modèle Nit sont transformées en des vecteurs multi-dimensionnels. Cette opération se fait grâce au module `vsm` (section 6.5), un module Nit pour la création de modèles d'espaces vectoriels — *Vector Space Model (VSM)* en anglais.

Chaque *dimension* du vecteur représente un attribut — au sens d'une information — sur l'entité Nit comme son nom, son module d'appartenance, les mots issus de son commentaire dans le code ou encore le nom de ses super-classes. À chaque dimension est assignée une *valeur*, une unité arbitraire utilisée pour pondérer l'importance de la dimension par rapport aux autres. Un exemple de vecteur représentant la classe `Array` est donné en figure 8.10.

La recherche par vecteur consiste alors à trouver les vecteurs dans l'index qui sont les plus similaires au vecteur de requête. Pour ce faire, nous utilisons la *similarité*

```
{
  "full_name: core::Array": 10,
  "name: Array": 5,
  "lemma: array": 2,
  "kind: MClass": 1,
  "is_example: false": 1,
  "is_test: false": 1,
  "parent: core::Sequence": 1,
  "comment: order": 1,
  "comment: sequence": 1,
  "comment: element: 1",
  ...
}
```

Figure 8.10: Représentation de la classe `Array` sous la forme d'un vecteur indexable.

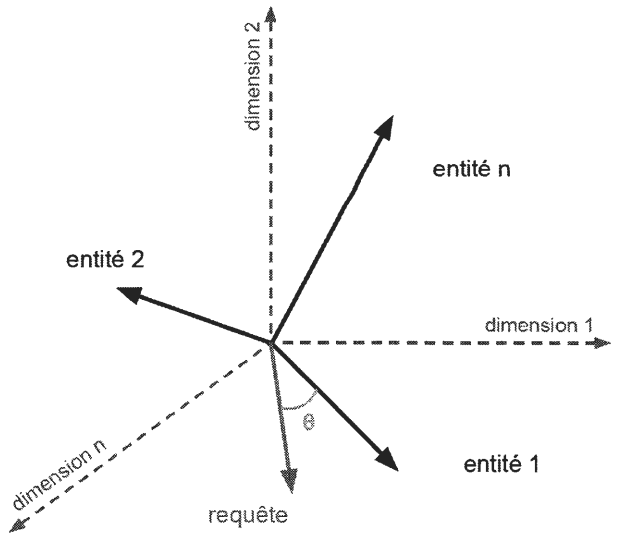


Figure 8.11: Représentation des entités Nit sous la forme de vecteurs.

cosinus (*cosine similarity*) représentant le cosinus de l'angle entre deux vecteurs multi-dimensionnels A et B, noté $cos\,\theta$ et défini comme suit :

$$cos\,\theta = \frac{A \cdot B}{||A|| \cdot ||B||}$$

Ici, $A \cdot B$ représente le produit scalaire des vecteurs A et B, alors que $||A|| \cdot ||B||$ représente le produit des normes des vecteurs. La valeur de $cos\,\theta$ est comprise entre 0.0 et 1.0, 0.0 indiquant deux vecteurs perpendiculaires, donc complètement différents, alors que 1.0 indique que les deux vecteurs sont colinéaires, donc semblables. Le vecteur **requête** de la figure 8.11 illustre cette similarité par rapport au vecteur pour `entité 1`, comparativement aux vecteurs des autres entités indiquées.

Pour trouver la liste des entités de l'index les plus pertinentes par rapport à un vecteur représentant une requête, on calcule donc la similarité cosinus du vecteur de requête avec chacun des vecteurs de l'index. On obtient ainsi la liste des entités les plus pertinentes que l'on peut trier selon la similarité $cos\,\theta$.

Par exemple, pour retrouver seulement la liste des classes qui ne sont pas des tests unitaires et contiennent les mots «*sequence*» ou «*array*» dans leur commentaire, on composera le vecteur suivant :

```
{
  "+kind: MClass": 1,
  "-is_test: true": 1,
  "comment: sequence": 1,
  "comment: array": 1,
  "sort: alpha": 1,
  "order: asc": 1,
  "limit": 5
  ...
}
```

Pour indiquer qu'une dimension est obligatoire, elle peut être préfixée par l'opérateur «+» dans la requête, comme le montre notre exemple pour la dimension `+kind: MClass`. Inversement, l'opérateur d'exclusion «-» permet d'indiquer que la dimension ne doit pas être présente comme pour `-is_test: true`.

Des dimensions spéciales sont aussi utilisables dans le vecteur de requête pour préciser les options de tri et de limite.

La dimension `sort` peut prendre une valeur qui spécifie l'ordre de tri :

- `similarity` : selon la similarité cosinus ;
- `alpha` : selon l'ordre lexicographique des noms courts des entités ;
- `visibility` : selon la visibilité, entités publiques puis protégées puis privées ;
- `kind` : selon le type, d'abord les packages puis les groupes, les modules, les classes et les propriétés ;
- `location` : selon l'ordre de déclaration dans le code ou l'ordre `alpha` pour les packages, groupes et modules ;
- `count(dimension)` : selon la valeur dans la dimension précisée entre parenthèses. Par exemple, `count(comment: array)` triera selon le nombre d'occurrences du lemme `array` dans le commentaire de l'entité.

Pour la dimension `order`, les valeurs `asc` ou `desc` indiquent un ordre ascendant ou descendant de tri. Quant à la dimension `limit`, elle permet de spécifier le nombre de résultats à retourner.

8.4.1 Indexation des noms des entités

L'indexation des noms permet de retrouver les entités du modèle Nit composant l'API à partir de leurs noms. Ce processus est illustré à la figure 8.12. Il débute en visitant le modèle Nit de l'API afin d'en extraire la liste des entités définies telles que les packages, les modules, les classes et les propriétés.

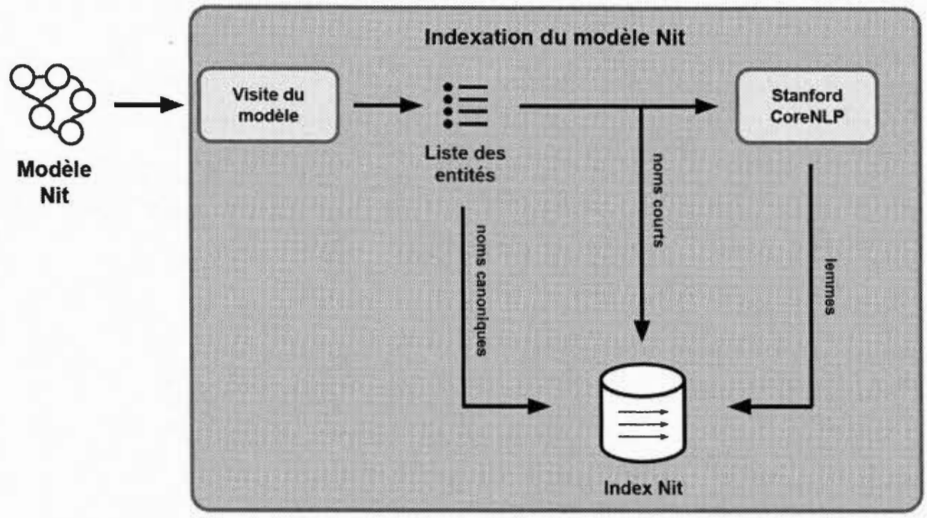


Figure 8.12: Processus d'indexation des noms canoniques, courts et lemmatisés des entités de l'API Nit.

Trois informations sont indexées à partir de cette liste :

1. Les *noms canoniques* associant chaque entité du modèle à son nom canonique. Exemple : `"core::ArraySet"` → *classe* ***`ArraySet`*** *du modèle Nit.*
2. Les *noms courts* associant chaque entité du modèle à son nom court. Exemple : `"ArraySet"` → *classe* ***`ArraySet`*** *du modèle Nit.*
3. Les *noms lemmatisés* associant chaque entité du modèle à la racine, ou lemme (en anglais *lemma*), de son nom court grâce au processeur de langue naturelle Stanford CoreNLP (Manning *et al.*, 2014). Exemple : `"array"` ou `"set"` → *classe* ***`ArraySet`*** *du modèle Nit.*

Lors des phases suivantes, les noms seront utilisés pour aligner les mises en relief de code Markdown (section 8.6.2) et les identifiants dans le texte (section 8.6.3) aux entités du modèle Nit. Les noms sont aussi utilisés pour l'alignement des extraits de code non compilables (section 8.6.4).

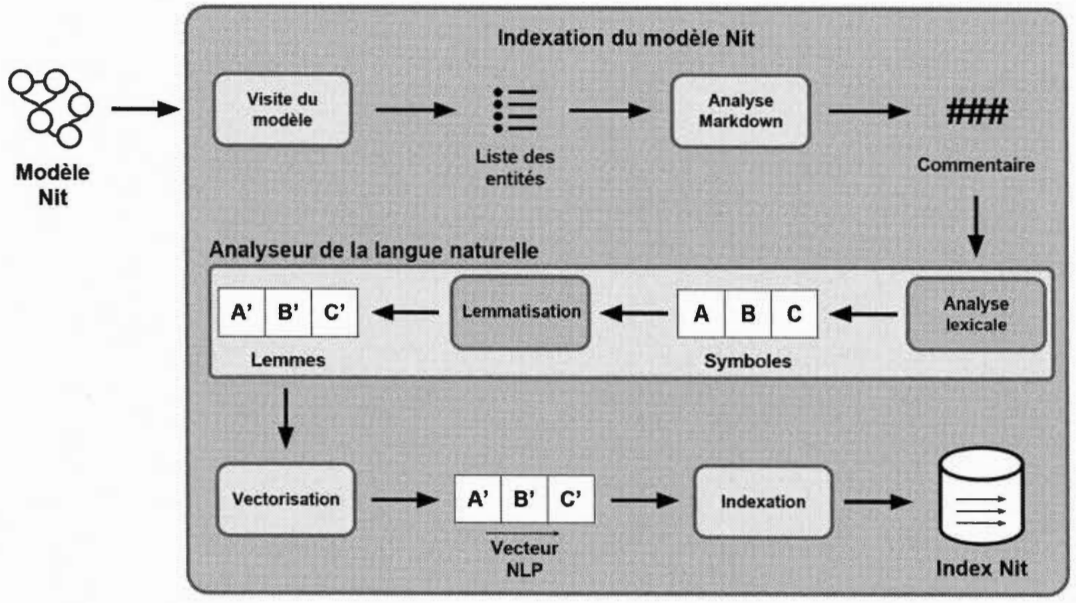


Figure 8.13: Processus d'indexation des entités du modèle par leurs commentaires.

8.4.2 Indexation des commentaires

L'indexation des commentaires permet de retrouver une entité du modèle Nit à partir d'un ensemble de termes exprimés en langage naturel — dans notre cas l'anglais — en comparant leur proximité avec les termes trouvés dans le commentaire de l'entité. La figure 8.13 donne un aperçu de ce processus d'indexation.

Cette information permettra plus tard d'aligner le contenu des paragraphes Markdown avec celui des commentaires des entités Nit (section 8.6.5).

Visite du modèle Pour chaque entité du modèle, son commentaire Markdown, tel qu'écrit dans le code source pour les modules, classes et propriétés ou dans le fichier README pour les packages (section 5.2), est récupéré.

Analyse Markdown Le commentaire de chaque entité est ensuite compilé par l'analyseur Markdown (voir `nitmd`, section 5.2) vers du texte brut. Cette opération

permet de supprimer les caractères liés à la mise en forme Markdown pour ne conserver que le texte du commentaire. Les extraits de code en ligne ou en blocs sont ensuite supprimés.

Lemmatisation Le commentaire brut ainsi obtenu est passé à l'outil d'analyse de langue naturelle afin d'être découpé en symboles (*tokens*). Chaque symbole est ensuite réduit en sa forme canonique ou *lemme*. Les verbes issus du commentaire sont transformés en leur forme à l'infinitif et les noms en leur forme au masculin singulier. Notre implémentation se base sur l'analyseur de langue naturelle Stanford CoreNLP (Manning *et al.*, 2014) et son connecteur Nit `nlp` (section 6.5).

Vectorisation Cette étape permet d'obtenir une représentation simplifiée de chaque commentaire, c'est-à-dire l'ensemble des lemmes utilisés pour documenter l'entité. Cette suite de lemmes est transformée en un vecteur multi-dimensionnels avant d'être indexée. Chaque dimension de ce vecteur représente le poids de chaque terme (lemme) composant le commentaire.

Nous avons utilisé la méthode de pondération *TF-IDF* pour déterminer le poids de chacun des termes (Salton *et al.*, 1975). Cette approche permet de mesurer l'importance de chaque terme contenu dans un document (ici un commentaire) par rapport au corpus de tous les documents (commentaires) à indexer. Ce poids augmente avec la fréquence du terme dans le document (commentaire) ainsi que la fréquence du terme dans l'ensemble du corpus.

L'approche *TF-IDF* est souvent utilisée en recherche d'information pour l'implémentation de moteurs de recherche plein texte. C'est d'ailleurs cette même approche qui est utilisée pour l'implémentation du moteur de recherche de `nitweb` (section 6.2).

Pour calculer les valeurs de *TF-IDF*, on commence par calculer l'importance de chaque terme t pour chaque document représenté par le commentaire de l'entité noté c. Cette importance est mesurée par le nombre d'occurrences de t dans c, appelée *fréquence* et notée $tf_{t,c}$ (*term frequency*).

La fréquence du terme se calcule grâce à la formule suivante :

$$tf_{t,c} = \frac{f_{t,c}}{\sum_{t' \in c} f_{t',c}}$$

Ici, $f_{t,c}$ représente le nombre d'occurrences de t dans c et $\sum_{t' \in c} f_{t',c}$ représente le nombre total de termes dans c. La valeur de $tf_{t,c}$ est comprise entre 0.0 et 1.0 : une valeur de 0.0 indique que t est absent du commentaire c, alors qu'une valeur égale à 1.0 indique que le commentaire ne contient que le terme t.

On calcule ensuite l'importance de chaque terme t par rapport à l'ensemble du corpus de commentaires à indexer. Cette mesure, appelée *fréquence inverse de document* (*inverse document frequency*), vise à donner un poids plus important aux termes les moins fréquents dans le corpus, et donc les plus discriminants de chaque commentaire. Elle se calcule selon la formule suivante :

$$idf_t = log \frac{|C|}{|\{c_i \in C : t \in c_i\}|}$$

Ici, $|C|$ représente le nombre total de commentaires dans le corpus alors que $|\{c_i \in C : t \in c_i\}|$ indique le nombre de commentaires où le terme t apparaît. Une valeur idf_t proche de 0.0 indique que le terme t n'est pas discriminant dans l'ensemble du corpus de commentaires — par exemple s'il se trouve dans tous les commentaires ; au contraire, plus idf_t augmente, plus le terme t est discriminant.

Finalement, le poids d'un terme t pour un commentaire c s'obtient en multipliant les deux mesures précédentes :

$$tfidf_{t,c} = tf_{t,c} \cdot idf_t$$

La valeur $tfidf_{t,c}$ est comprise entre 0.0 et 1.0, 0.0 indiquant que le terme t n'est pas discriminant pour le commentaire c, 1.0 indiquant que le terme est fortement discriminant.

Indexation Le vecteur final représentant le commentaire c est obtenu en associant la valeur de $tfidf_{t,c}$ à la dimension représentée par chaque terme t. Ce vecteur est ensuite ajouté à l'index permettant ainsi de récupérer une entité grâce à son vecteur.

À titre d'exemple, prenons les trois classes suivantes accompagnées des commentaires indiqués :

```
core::Array -- One dimension array of objects
core::Range -- Range of discrete objects
markdown::HtmlRenderer -- Translation of Markdown strings to HTML strings
```

Le tableau 8.2 donne le détail du calcul de *TF-IDF* pour chaque terme dans chaque commentaire. On remarque que le terme « *of* » est présent dans chacun des commentaires et obtient ainsi un poids nul car il n'est pas discriminant. Le terme « *object* » est présent dans deux commentaires, il obtient donc un poids plus faible que les termes présents dans un seul commentaire. En revanche, le terme « *string* » est particulièrement représentatif du commentaire de `markdown::HtmlRenderer` car il apparaît deux fois dans son commentaire et seulement dans ce commentaire.

L'approche *TF-IDF* présente l'avantage de diminuer automatiquement le poids des termes courants, donc apparaissant dans de nombreux commentaires, tels que les articles ou adjectifs. Cette pondération à la baisse est automatiquement ajustée en fonction du corpus et du nombre d'occurrences de chaque terme, ce qui évite d'avoir à maintenir une liste exhaustive des termes à filtrer.

8.4.3 Indexation du code source, des exemples et des tests

L'indexation du code source permet de retrouver des entités du modèle Nit grâce à une représentation abstraite de leur code source, par exemple, retrouver la liste des entités qui importent le module X, instancient la classe Y et appellent la méthode Z. Le processus d'indexation du code est illustré à la figure 8.14.

(a) Commentaire de `core::Array`

Terme	**Lemme**	$f_{t,c}$	$tf_{t,c}$	idf_t	$tfidf_{t,c}$
`One`	one	1	0.2	0.5	0.1
`dimension`	dimension	1	0.2	0.5	0.1
`array`	array	1	0.2	0.5	0.1
`of`	of	1	0.2	0.0	0.0
`objects`	object	1	0.2	0.2	0.0

(b) Commentaire de `core::Range`

Terme	**Lemme**	$f_{t,c}$	$tf_{t,c}$	idf_t	$tfidf_{t,c}$
`Range`	range	1	0.3	0.5	0.1
`of`	of	1	0.3	0.0	0.0
`discrete`	discrete	1	0.3	0.5	0.1
`objects`	object	1	0.3	0.2	0.0

(c) Commentaire de `markdown::HtmlRenderer`

Terme	**Lemme**	$f_{t,c}$	$tf_{t,c}$	idf_t	$tfidf_{t,c}$
`Translation`	translation	1	0.1	0.5	0.1
`of`	of	1	0.1	0.0	0.0
`Markdown`	Markdown	1	0.1	0.5	0.1
`strings`	string	2	0.3	0.5	0.1
`to`	to	1	0.1	0.5	0.1
`HTML`	HTML	1	0.1	0.5	0.1

Tableau 8.2: Calcul de la fréquence de chaque terme, de la fréquence de document inverse et de *TF-IDF* pour les trois commentaires de notre exemple.

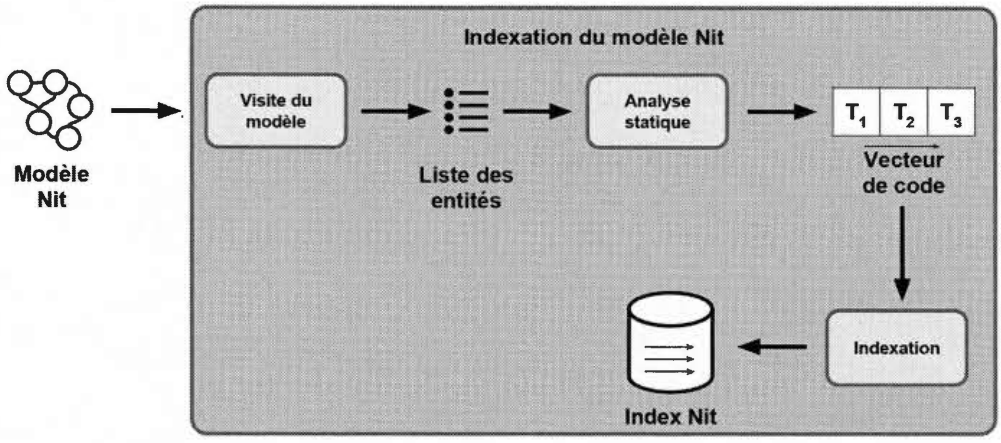


Figure 8.14: Processus d'indexation des extraits de code.

Cette information sera utilisée lors de l'alignement pour trouver les entités Nit dont l'implémentation est similaire à un extrait de code Markdown contenant du Nit (section 8.6.4).

La recherche d'extraits de code à partir de la langue naturelle est déjà un sujet bien traité dans la littérature scientifique. Bajracharya *et al.* (2006) et Sindhgatta (2006), pour ne citer qu'eux, proposent une approche basée sur *TF-IDF* afin de faire coïncider des requêtes en langue naturelle avec des extraits de code Java.

Puisque nous utilisons déjà *TF-IDF* pour la transformation des commentaires en vecteurs (section 8.4.2), nous avons choisi d'utiliser une approche similaire pour la création de l'index de code. Ainsi, nous avons repris les travaux de Bajracharya *et al.* (2006) et les avons adaptés aux besoins de la suggestion de documentation.

Le vecteur d'un extrait de code est composé de manière à représenter au mieux l'utilisation faite des modules, types et propriétés. Nous cherchons donc à représenter les entités utilisées telles les packages, modules, classes et propriétés mais aussi la façon dont ces entités sont utilisées. Par exemple, est-ce qu'un nom de classe est utilisé comme type dans une signature ou pour appeler un constructeur et donc en créer une instance ?

Tableau 8.3: Descripteurs d'actions utilisés dans les vecteurs de code.

Action	**Descripteur**	**Exemple**
Module importé.	`import`	`import: markdown`
Classe utilisée comme superclasse.	`super`	`super: markdown::Decorator`
Classe utilisée comme type de paramètre dans la signature d'une méthode.	`param`	`param: core::String`
Classe utilisée comme type de retour dans la signature d'une méthode.	`return`	`return: core::Int`
Classe utilisée comme borne de type virtuel dans une classe.	`vtype`	`vtype: core::Object`
Classe utilisée comme borne de type formel dans la définition d'une classe générique.	`ftype`	`ftype: core::Object`
Classe utilisée dans la création d'une instance.	`new`	`init: core::Array`
Classe utilisée pour typer une variable locale.	`var`	`var: core::Int`
Propriété appelée (méthode, constructeur ou accesseur d'attribut).	`call`	`call: core::String::split`
Nom canonique utilisé.	`name`	`full_name: core::Object`
Nom court utilisé.	`name`	`name: Object`

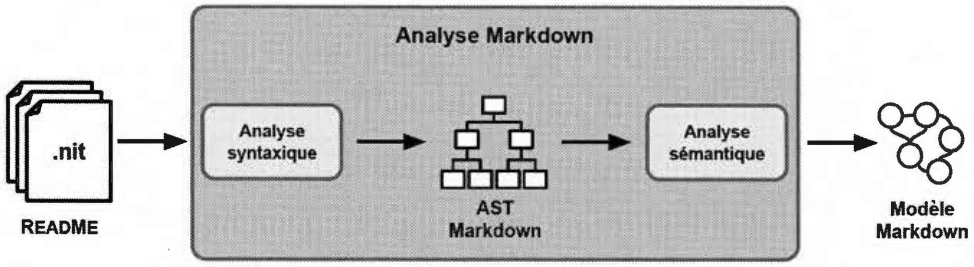


Figure 8.15: Processus d'analyse du fichier README en cours d'écriture pour la génération du modèle Markdown.

Ainsi, le vecteur représentant un extrait de code se compose sur plusieurs dimensions représentant à la fois les entités utilisées et le type d'usage. Pour ce faire, chaque dimension est représentée par le nom canonique de l'entité utilisée préfixée d'un *descripteur d'action*, lequel indique le rôle que joue l'entité dans l'extrait de code. Le tableau 8.3 liste les descripteurs d'actions ajoutés aux dimensions du vecteur en fonction de l'utilisation faite de l'entité. La valeur associée à chaque dimension est représentée par le nombre d'occurrences de l'action liée dans l'extrait de code analysé.

Pour chaque entité dans le modèle Nit, les nœuds de l'arbre syntaxique abstrait (AST) liés à son code source sont récupérés. Cet AST est ensuite traité par une analyse statique du code source pour composer le contenu du vecteur. Cette analyse est implémentée à l'aide du cadre de développement pour l'analyse statique de code Nit nommé `saf` (section 6.5).

Une fois le vecteur créé, il est ajouté à l'index Nit permettant de retrouver l'entité Nit grâce à son vecteur de code.

8.5 Analyse du fichier README et création du modèle Markdown

Le processus d'analyse du fichier README en cours d'écriture (*en ligne*), présenté à la figure 8.15, permet la création d'un modèle du contenu Markdown qui peut ensuite être aligné avec les entités du modèle Nit.

Le processus commence tout d'abord en analysant le contenu du fichier Markdown pour créer un arbre syntaxique grâce à l'outil `nitmd` (chap. 5.2). Cet arbre offre une représentation abstraite du contenu Markdown et permet d'en parcourir le contenu facilement.

À partir de cet arbre, on construit le modèle du fichier représentant sa structure et son contenu. Nous appelons cette représentation abstraite le «*modèle Markdown*». Cette représentation se veut de plus haut niveau que l'arbre syntaxique et simplifie l'implémentation des analyses subséquentes.

Le modèle Markdown permet de naviguer dans le document selon la hiérarchie des sections et des blocs, tel qu'illustré dans la figure 8.16. Il contient la liste et l'imbrication des blocs Markdown (sections, listes, paragraphes, etc.) et les liens entre ces blocs. Il contient aussi, pour chaque bloc, un contexte qui sera enrichi par les informations générées lors du processus d'alignement, que nous présentons dans les sections suivantes.

8.6 Alignement du contenu du fichier README avec les entités Nit et validation

L'alignement du contenu du fichier README se fait lui aussi lors de l'étape *en ligne*, au cours de sa rédaction par l'écrivain. Le contenu Markdown extrait lors de l'analyse du fichier est alors aligné avec les entités du modèle grâce à l'index créé lors de l'étape *hors ligne*.

8.6.1 Alignement des thèmes

Les thèmes utilisés dans notre analyse des fichiers README provenant de GitHub (chap. 2) permettent de comprendre l'intention des sections et des blocs présents

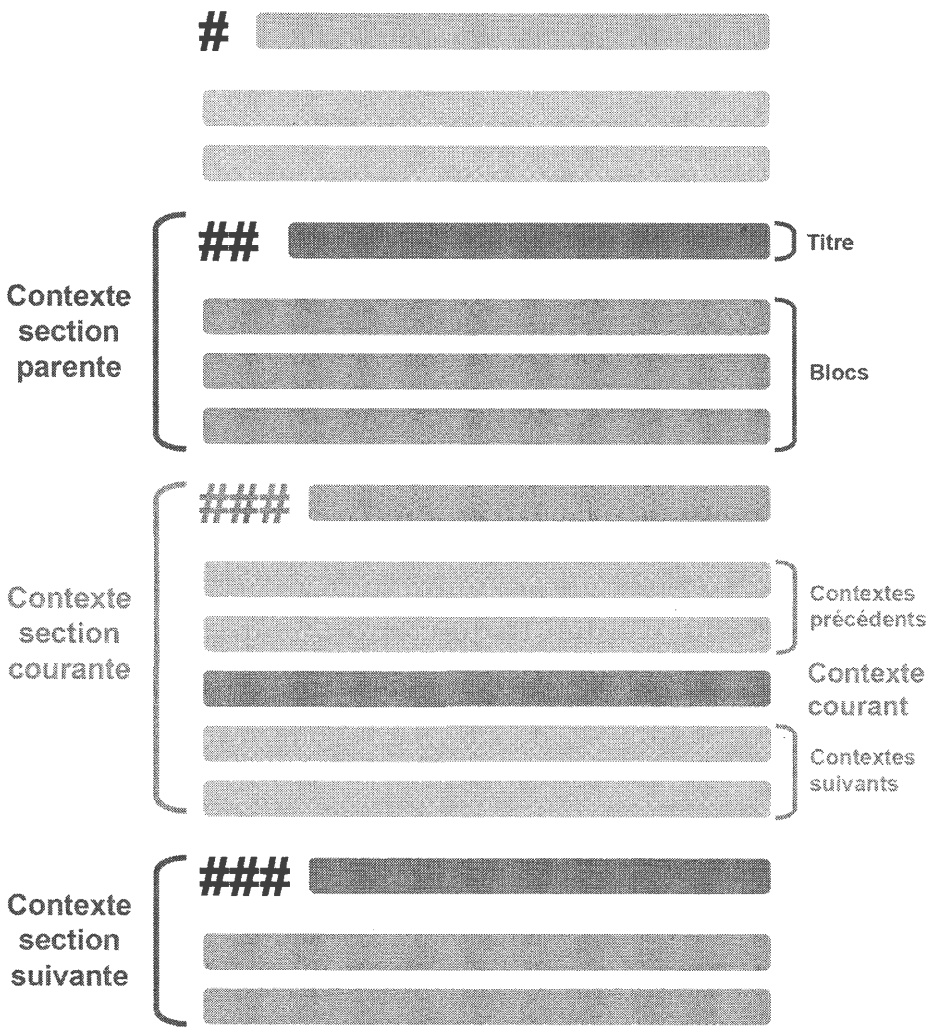


Figure 8.16: Illustration des contextes dans le modèle du document.

dans le document Markdown, par exemple, déterminer si une section parle de l'installation, d'un élément de l'API ou de la licence.

Ces thèmes sont utilisés pour filtrer les résultats obtenus lors de l'alignement des identifiants trouvés dans le texte (section 8.6.3) et de l'alignement de la langue naturelle (section 8.6.5) ainsi que pour déterminer les cartes de documentation à insérer (chap. 9).

8.6.1.1 Alignement des thèmes contenus dans le README

Pour chaque bloc Markdown présent dans le document, la phase d'alignement analyse le contenu du bloc afin de déterminer les thèmes les plus plausibles. Nous avons ajouté cinq thèmes à la liste originale utilisée au chapitre 2 permettant de mieux déterminer la cible de la documentation. Ces thèmes sont basés sur les types d'entités disponibles en Nit :

- *Package* : `/package/`, `/library/` ;
- *Groupe* : `/group/`, `/folder/`, `/directory/` ;
- *Module* : `/module/` ;
- *Classe* : `/class/`, `/interface/` ;
- *Propriété* : `/property/`, `/method/`, `/attribute/`, `/constructor/`.

En-têtes de sections Le sections sont alignées grâce au texte contenu dans leurs en-têtes de sections Markdown. Le texte des en-têtes est lemmatisé grâce au processeur de langue naturelle Stanford CoreNLP (Manning *et al.*, 2014). Des expressions régulières sur les lemmes de l'en-tête sont ensuite utilisées afin de déterminer le thème de la section. Par exemple, l'expression régulière `/license/` permet d'aligner les en-têtes contenant les chaînes `"License"` ou `"Licensing"` comme appartenant au thème *Licence* une fois ces chaînes lemmatisées.

Paragraphes Les paragraphes sont alignés suivant la même stratégie que les en-têtes de sections mais avec le texte du paragraphe. Par contre, une phase récursive est ajoutée, laquelle permet d'importer au niveau du paragraphe les thèmes trouvés dans les éléments de la syntaxe en ligne Markdown. Ainsi, les thèmes associés aux liens, images, mises en relief de texte ou de code trouvés dans les paragraphes sont importés au niveau du paragraphe.

Listes ordonnées et non-ordonnées Les thèmes des listes ordonnées sont sélectionnés en fonction de leur contenu. Une liste ordonnée contenant des liens internes au document est considérée comme un *Sommaire*. Une liste ordonnée présentant des mises en relief de code est associé au thème *Utilisation* si les mises en relief de code font références à des entités Nit, ou au thème *Installation* si les mises en relief de code font références à des commandes Shell.

Pour les thèmes des listes non-ordonnées, les mises en relief de code permettent d'associer la liste au thème *API*. Les liens vers la documentation d'API permettent d'associer la liste au thème *Documentation*. Les autres liens externes ajoutent le thème *Références*. La présence de noms et prénoms ajoutent le thème *Auteurs*.

Extraits de code Les extraits de code sont alignés en fonction du langage de programmation utilisé et du contenu. Les extraits déclarés comme du Nit sont automatiquement associés au thème *Exemple*. Les extraits déclarés comme du Shell sont automatiquement associés au thème *Usage* s'ils contiennent une référence à un outil Nit (expression régulière `/nit[a-z]*/`) ou le nom d'un des exécutables trouvés dans le package. Les autres extraits de code Shell sont associés au thème *Installation*.

8.6.1.2 Validation de l'alignement des thèmes

Nous avons annoté manuellement le corpus de fichiers README afin d'associer les sections et blocs Markdown utilisés aux thèmes auxquels ils correspondent. Le corpus une fois annoté permet de comparer les thèmes assignés par notre algorithme par rapport aux thèmes utilisés par les écrivains. Il est alors possible de calculer le *rappel* et la *précision* de notre algorithme sur le corpus de fichiers README.

Le *rappel*, qui représente la proportion de thèmes sélectionnées par l'algorithme d'alignement parmi tous les thèmes possibles, est calculé comme suit :

$$rappel = \frac{|\{thèmes\ pertinents\} \cap \{thèmes\ sélectionnés\}|}{|\{thèmes\ sélectionnés\}|}$$

Un rappel de 1.0 indique que tous les thèmes pertinents ont été assignés au bloc, alors qu'un rappel de 0.0 indique un *silence*, c'est-à-dire, aucun des thèmes pertinents n'a été sélectionné.

La *précision*, qui représente la proportion de thèmes pertinents parmi tous les thèmes sélectionnés par l'algorithme d'alignement, est calculée comme suit :

$$précision = \frac{|\{thèmes\ pertinents\} \cap \{thèmes\ sélectionnés\}|}{|\{thèmes\ pertinents\}|}$$

Une précision de 1.0 indique que tous les thèmes sélectionnés sont pertinents, alors qu'une précision de 0.0 indique que l'algorithme ne produit que du *bruit*.

Le tableau 8.4 présente le rappel de notre algorithme d'alignement des thèmes pour chaque package du corpus. Le rappel moyen est de 0.97, la majorité des blocs sont donc alignés correctement avec thèmes utilisés. La précision moyenne est de 0.94, donc certains blocs sont associés aux mauvais thèmes.

Tableau 8.4: Rappel et précision de l'algorithme de sélection des thèmes utilisés dans les blocs Markdown.

Bibliothèque	**Rappel**	**Précision**
`actors`	0.98	0.96
`ai`	1.00	1.00
`android`	0.96	0.90
`app`	0.95	0.90
`gamnit`	1.00	0.96
`geometry`	0.97	0.97
`github`	0.91	0.96
`ios`	0.92	0.88
`json`	0.98	0.98
`markdown`	1.00	1.00
`nitcorn`	1.00	0.88
`nlp`	0.98	0.89
`popcorn`	0.93	0.91
`posix`	1.00	1.00
`pthreads`	0.88	0.81
`sdl2`	1.00	1.00
`serialization`	0.96	0.93
`vsm`	1.00	0.96
Moyenne	**0.97**	**0.94**

Une revue manuelle des résultats nous a permis de constater que c'est le thème *API* qui est trop souvent associé à tort. En effet, les fichiers README du corpus Nit font souvent référence au code, et ce même quand le bloc ne traite pas explicitement d'un élément de l'API.

Un exemple d'une telle situation se trouve dans le README du package `nitcorn`, un cadre de développement pour des serveurs web en Nit. Dans la section sur les auteurs, on peut y lire :

```
This `nitcorn` library is a fork from an independent project
originally created in 2013 by Jean-Philippe Caissy,
Guillaume Auger, Frederic Sevillano, Justin Michaud-
Ouellette, Stephan Michaud and Maxime Bélanger.
```

Notre algorithme associe deux thèmes à ce paragraphe : *API* à cause de la présence de la mise en relief de code `` `nitcorn` `` faisant référence au nom du package et *Auteurs* pour la présence des noms et des prénoms.

Cependant, et selon l'utilisation que nous ferons des thèmes au chapitre suivant (chap. 9), une précision de 0.94 nous parait suffisante.

8.6.2 Alignement des mises en relief de code

Comme nous l'avons observé dans la section 8.2.3, les mises en relief de code représentent l'élément Markdown le plus fréquent dans les fichiers README de notre corpus avec 422 occurrences. Ces références représentent donc un élément d'information de choix pour aligner le contenu du README avec celui de l'API.

Cet alignement permet de déterminer quelles sont les entités Nit auquel l'écrivain fait référence dans le Markdown du fichier README. Par exemple, il sera utilisé pour la suggestions des cartes de documentation d'API telles les cartes de liens ou de commentaires.

8.6.2.1 Alignement avec l'index Nit

L'alignement des noms trouvés dans les mises en relief de code se fait grâce à l'index Nit construit lors de l'analyse du code source Nit (phase *hors ligne* : section 8.4.1).

Les noms qualifiés comme `"core::Array"` ou `"core::Array::join"` sont directement alignés avec les entités contenues dans l'API grâce à la dimension `full_name` dans l'index Nit. Dans le cas où le nom qualifié correspond à une méthode et contient une partie de la signature comme `"core::Array::join(string)"` ou bien un type générique sur une classe comme `"core::Array[Int]"`, la signature est supprimée afin de ne conserver que le nom. Les résultats obtenus par l'alignement des noms qualifiés ne nécessitent pas de filtrage particulier puisque ces noms ne peuvent pas produire de conflits (Subramanian *et al.*, 2014).

Les noms semi-qualifiés tels `"Array::join"` sont d'abord alignés en fonction du nom de la première entité grâce à la dimension `name` dans l'index Nit — dans notre exemple `"Array"` — puis sont filtrés en fonction de l'existence des entités suivantes — ici `"join"`. Ainsi l'algorithme limite les conflits dans le cas où une seconde classe nommée `Array` existerait mais sans propriété `join`.

Les noms courts comme `"Array"` ou `"join"` sont eux aussi alignés grâce à la dimension `name`. L'algorithme sélectionne toutes les entités du même nom quel que soit le type de l'entité (package, module, classe, propriété) ou le package dont elle provient.

Comme c'est la cas dans l'approche proposée par Dagenais et Robillard (2012), nous utilisons les signatures pour désambiguïser les noms courts et les noms semi-qualifiés. Ainsi, si la signature indique `"join(string)"`, alors l'algorithme désambiguïse en fonction du nombre de paramètres définis dans l'introduction de la méthode. De la même manière, les types génériques sont utilisés pour désambiguïser

les noms de classes. Ainsi, `"Array[Int]"` permet de sélectionner la classe générique `Array[E]` au lieu d'une éventuelle classe du même nom avec un nombre différent de types formels.

L'alignement des noms courts introduit du bruit dans les résultats à cause des conflits de noms qui peuvent survenir. En effet, plusieurs classes dans des packages différents peuvent porter le même nom court, et il en est de même pour les propriétés dans des classes différentes. Sans information sur la signature, il est difficile de savoir quelle est l'entité attendue par l'écrivain. Dagenais et Robillard (2012) parlent dans ce cas d'une ambiguïté de déclaration (*declaration ambiguity*) et précisent que le lecteur peut en général désambiguïser ces références grâce au contexte. Dans leur approche, ils utilisent le contexte du document pour filtrer les résultats obtenus. Ce contexte dépend de la position dans le document, des sections parentes, des sous-sections et du contexte global du document.

Dans le cas des fichiers README, le contexte est plus facile à déterminer car il dépend avant tout du package en cours de documentation par l'écrivain. Nous avons choisi de résoudre les conflits de noms en filtrant les entités en fonction de leur appartenance au package relatif au fichier README.

L'alignement des noms courts ajoute un second type de conflit en fonction du type des entités. En Nit, les packages, groupes et modules peuvent porter le même nom court ; or, les écrivains font en général référence au package plutôt qu'aux groupes et aux modules. Afin de limiter le bruit, nous avons choisi de filtrer systématiquement les groupes et modules si un package du même nom est présent dans les résultats de l'alignement.

8.6.2.2 Validation de l'alignement des mises en relief de code

Notre approche de validation est semblable à celles utilisées par d'autres chercheurs (Antoniol *et al.*, 2002; Dagenais et Robillard, 2012; Rigby et Robillard, 2013). Comme pour l'alignement des thèmes, nous avons annoté manuellement le corpus de fichiers README afin d'associer les mises en relief de code utilisées aux entités de l'API qu'elles sont supposées référencer. Le corpus une fois annoté permet de comparer les entités détectées par notre algorithme par rapport aux entités attendues par l'écrivain. Les définitions de rappel et précision sont les mêmes que précédemment, mais cette fois par rapport aux entités plutôt qu'aux thèmes.

Le tableau 8.5 présente le rappel de notre algorithme pour chaque package du corpus (`nlp` ne contenait aucune mise en relief). Le rappel moyen étant de 0.98, la majorité des entités sont donc sélectionnées correctement. Notre rappel moyen est supérieur à celui obtenu par les autres approches citées dans les travaux connexes où le maximum est celui de l'approche de Dagenais et Robillard (2012) à 0.96.

La colonne **P. base** indique la précision par défaut de l'algorithme sans aucune option de filtre. La précision moyenne est de 0.74, donc sans filtre les résultats contiennent donc beaucoup de bruit. Un filtre appliqué sur le contexte du document README (colonne **P. contexte**), c'est-à-dire sur le package en cours de documentation par l'écrivain, permet de limiter le bruit et augmente la précision à 0.78. Un filtre appliqué sur le type des entités sélectionnées (colonne **P. type**), c'est-à-dire favorisant la sélection des entités les plus générales, permet de limiter le bruit et augmente la précision à 0.92.

Les meilleurs résultats sont obtenus en combinant les deux filtres (colonne **P. tous**) avec une précision de 0.97, et donc avec un minimum de bruit.

Ces résultats sont semblables à ceux obtenus par Dagenais et Robillard (2012), qui atteignent une précision moyenne de 0.96. Les autres approches citées dans les travaux connexes sont généralement moins précises avec des moyennes inférieures à 0.92.

Tableau 8.5: Performances de l'algorithme de sélection des entités de l'API depuis les références des mises en relief de code.

Légende: **R.** et **P.** représentent respectivement le rappel et la précision de l'algorithme ; **base** : algorithme sans filtrage ; **type** : algorithme avec un filtrage sur le type ; **contexte** : algorithme avec un filtrage sur le package d'origine ; **tous** : précision de l'algorithme avec un filtrage sur le contexte et sur le type d'entité.

	base		type		contexte		tous	
Bibliothèque	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**
actors	1.00	0.47	1.00	0.69	1.00	0.62	1.00	0.84
ai	1.00	0.75	1.00	1.00	1.00	0.83	1.00	1.00
android	1.00	0.58	1.00	1.00	1.00	0.58	1.00	1.00
app	1.00	0.88	1.00	0.95	1.00	0.94	1.00	1.00
gamnit	1.00	0.53	1.00	0.71	1.00	0.69	1.00	0.95
geometry	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
github	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ios	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
json	1.00	0.94	1.00	1.00	1.00	0.94	1.00	1.00
markdown	1.00	0.42	1.00	0.75	1.00	0.67	1.00	1.00
nitcorn	0.88	0.89	0.88	1.00	0.88	0.89	0.88	1.00
nlp	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
popcorn	0.96	0.88	0.95	0.89	0.96	0.96	0.96	0.96
posix	1.00	0.50	1.00	1.00	1.00	0.50	1.00	1.00
pthreads	1.00	0.79	1.00	0.79	1.00	0.83	1.00	0.83
sdl2	1.00	0.55	1.00	0.88	1.00	0.55	1.00	0.88
serialization	0.85	1.00	0.85	1.00	0.85	1.00	0.85	1.00
vsm	1.00	0.33	1.00	1.00	1.00	0.33	1.00	1.00
Moyenne	**0.98**	**0.74**	**0.98**	**0.92**	**0.98**	**0.78**	**0.98**	**0.97**

8.6.3 Alignement des identifiants trouvés dans le texte

Une revue manuelle nous a permis d'isoler un total de 673 références dans le texte qui correspondent à des noms d'entités de l'API mais qui ne sont pas dans des mises en relief de code. Les noms présentés dans le texte du corpus Nit sont toujours des noms courts, sans signature ni type générique.

Plusieurs inflexions autour des noms sont possibles :

— Modification de la casse de caractères — par ex., `core::Array` vs. `"array"` ;

— Pluralisation ou singularisation du nom — par ex., `Array` vs. `"Arrays"` ;

— Conjugaison du nom — par ex., `cache` vs. `"caching"`.

Cet alignement permet de déterminer quelles sont les entités Nit auquel l'écrivain fait référence dans le texte du fichier README là où il n'a pas utilisé de mise en relief de code. Par exemple, il sera utilisé pour la suggestions des cartes de documentation d'API telles les cartes de liens ou de commentaires.

8.6.3.1 Alignement avec l'index Nit

Les noms des entités dans le texte sont isolés grâce à un *island parser* (Deursen et Kuipers, 1999), un analyseur syntaxique pouvant traiter des extraits de code *non compilables* (Deursen et Kuipers, 1999). Ce type de *parser* est basé sur une grammaire décrivant un sous-ensemble du langage à analyser et offre une analyse limitée des extraits de code. Le notre est construit selon un ensemble d'expressions régulières spécifique à la syntaxe des identifiants du langage Nit.

Chaque extrait de texte du fichier README, tel qu'un paragraphe ou un en-tête de section, est analysé à l'aide de l'*island parser* afin d'en extraire les identifiants présents dans le texte qu'il contient. Ces identifiants sont tout d'abord recherchés dans l'index grâce aux dimensions `full_name` et `name`. L'identifiant est ensuite

lemmatisé, toujours à l'aide du processeur de la langue naturelle Stanford CoreNLP, puis recherché dans l'index selon la dimension `lemma`.

Cette approche permet d'identifier les mots du texte faisant référence à des noms d'entités. Cependant, l'écrivain peut utiliser certains mots qui correspondent à des entités de l'API alors que le texte n'est pas en rapport avec ces entités. Par exemple, les mots `"string"`, `"value"` ou `"instance"` peuvent apparaître dans un document sans qu'il soit en rapport avec les entités `Regex::string`, `Node::value` ou `Actor::instance`. Ce genre d'ambiguïté peut produire une certaine quantité de faux positifs dans les résultats.

Comme le montrent Dagenais et Robillard (2012), le contexte du document permet généralement au lecteur de faire le tri entre les noms faisant référence au code et les noms sans rapport. Ils utilisent le contexte où apparaît le nom pour filtrer les entités homonymes. Ce contexte est basé sur le paragraphe contenant le nom, les paragraphes précédents et suivants, ainsi que les sections parentes jusqu'au document dans son ensemble.

Dans notre cas, les propriétés homonymes peuvent être filtrées selon le package en cours de documentation comme nous l'avons fait pour les mises en relief de code. Notre problème se situe plus sur les noms introduisant des entités sans conflit de noms mais sans rapport avec le texte rédigé par l'écrivain.

Un filtre brut sur les entités appartenant au package semble trop restrictif pour notre utilisation. En effet, nous souhaitons pouvoir aligner le contenu du README même si celui-ci traite d'entités externes au package, par exemple, pour faire référence à la bibliothèque standard ou à des packages connexes. Nous avonc donc choisi de réutiliser la notion de contexte d'apparition proposée par Dagenais pour l'appliquer sur les entités sans conflits de nom.

Pour chaque entité trouvée lors de l'alignement qui n'appartient pas au package en cours de documentation, notre filtre considère les paragraphes de la même

section afin d'y trouver une entité parente. Par exemple, pour le texte `"value"`, si l'alignement retourne la propriété `Node::value`, on cherche dans les paragraphes de la section si la classe `Node` est nommé que ce soit par une mise en relief de code, un extrait d'exemple ou un autre identifiant dans le texte. Si les paragraphes voisins contiennent le parent `Node`, alors l'entité `Node::value` est conservée, sinon elle est exclue des résultats. Si aucune entité parent n'est trouvée dans la section courante, on répète le processus avec la section parente, et ainsi de suite jusqu'à la racine du document.

Ce processus permet ainsi de filtrer les noms de propriétés grâce à l'apparition des classes, les classes grâce à l'apparition des modules, les modules grâce à l'apparition des groupes et des packages, et les noms des groupes grâce à l'apparition des packages. Les packages, généralement moins nombreux et plus rarement source de bruit, sont toujours conservés.

Certaines entités sont parfois référencées par leur nom sans pour autant que leur parent soit explicité dans le texte. Afin d'éviter le filtrage de ces entités, nous proposons une seconde variation du filtre de Dagenais mais prenant en compte le thème du bloc Markdown contenant l'identifiant. Les thèmes sont extraits lors de la phase d'alignement des thèmes — phase présentée à la section 8.6.1.

En plus du contexte du document général, des sections parentes et des voisins, le filtre utilise les thèmes associés aux blocs Markdown afin de déterminer les entités les plus intéressantes pour le contexte. Ceci se fait d'abord en fonction du type d'entité à filtrer grâce aux thèmes *Package*, *Group*, *Module*, *Class* et *Property*, puis grâce à des heuristiques en fonction du thème du bloc :

- *Titre* : Conserve uniquement le package en cours de documentation ;
- *Introduction*, *Objectif*, *Installation* : Conserve uniquement les références aux groupes, modules et classes se trouvant dans le contexte du package en cours de documentation et conserve tous les packages ;

— *Utilisation* : Conserve toutes les entités annotées comme des exemples ainsi que toutes les entités se trouvant dans le contexte du package courant ;
— *API, Fonctionnalités* : Conserve tout ;
— *Utilisation* : Conserve toutes les entités annotées comme des exemples ainsi que toutes les entités se trouvant dans le contexte du package courant ;
— *Tests* : Conserve toutes les entités annotées comme des tests unitaires ainsi que toutes les entités se trouvant dans le contexte du package courant ;
— *Aide, Contribuer, Licence* : ne conserve rien.

8.6.3.2 Validation de l'alignement des noms des entités

Nous avons annoté manuellement le corpus de fichiers README afin d'associer les noms présents dans le texte aux entités de l'API qu'ils sont supposés référencer. Le corpus une fois annoté permet de comparer les entités détectées par notre algorithme par rapport aux entités attendues par l'écrivain. Le tableau 8.6 présente le rappel et la précision de notre algorithme pour chaque package du corpus.

Les colonnes **R. base** et **P. base** indiquent respectivement le rappel et la précision de l'algorithme seulement sur les noms exacts et sans aucune option de filtre. Le rappel moyen est de 0.47, moins de la moitié des références sont donc faites grâce aux noms exacts. La précision moyenne est de 0.48, sans filtre les résultats contiennent déjà beaucoup de bruit — plus de la moité des résultats retournés sont incorrects.

L'utilisation des lemmes des noms permet d'atteindre un rappel de 0.98 (colonne **lemmes R.**). Ce rappel est beaucoup plus acceptable que celui sans l'utilisation des lemmes. En revanche la précision descend à 0.40 (colonne **lemmes P.**).

Le filtre sur le contexte adapté de Dagenais et Robillard (2012) permet d'atteindre une précision de 0.86 (colonne **Dagenais P.**). Cependant, le contexte global du

Tableau 8.6: Performances de l'algorithme d'alignement grâce aux identifiants dans le texte.

Légende: **R.** et **P.** représentent respectivement le rappel et la précision de l'algorithme; **base** : alignement sur les noms exacts; **lemmes** : alignement sur les noms exacts et les lemmes; **Dagenais** : filtrage sur le contexte adapté depuis l'article de Dagenais et Robillard (2012); **thème** : filtrage sur le contexte de Dagenais et adapté à nouveau pour prendre en compte les thèmes.

	base		lemmes		Dagenais		thèmes	
Bibliothèque	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**
`actors`	0.14	0.11	1.00	0.22	0.94	0.74	1.00	0.68
`ai`	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
`android`	0.27	0.44	1.00	0.18	0.79	0.70	0.96	0.62
`app`	0.49	0.24	1.00	0.27	0.69	0.86	1.00	0.80
`gamnit`	0.63	0.19	1.00	0.17	0.96	0.81	1.00	0.77
`geometry`	0.34	0.59	1.00	0.52	0.97	0.80	1.00	0.78
`github`	0.17	0.43	0.78	0.24	0.64	0.80	0.63	0.75
`ios`	0.55	0.17	1.00	0.20	0.90	0.93	1.00	0.85
`json`	0.11	0.75	0.94	0.31	0.75	0.82	0.94	0.77
`markdown`	0.89	0.50	1.00	0.42	0.93	1.00	1.00	0.86
`nitcorn`	0.77	0.23	1.00	0.28	0.90	0.91	1.00	0.84
`nlp`	0.70	0.86	1.00	0.87	0.80	0.93	0.95	0.92
`popcorn`	0.44	0.32	0.99	0.28	0.76	0.65	0.97	0.75
`posix`	0.00	0.50	1.00	0.25	1.00	0.83	1.00	0.83
`pthreads`	0.06	0.17	1.00	0.31	0.75	0.88	1.00	0.81
`sdl2`	0.78	0.86	0.87	0.78	0.84	0.88	0.87	0.78
`serialization`	0.67	0.55	1.00	0.39	0.83	0.99	1.00	0.87
`vsm`	0.52	0.67	0.98	0.56	0.84	0.92	0.98	0.88
Moyenne	**0.47**	**0.48**	**0.98**	**0.40**	**0.85**	**0.86**	**0.96**	**0.81**

document étant déjà fixé sur celui du package en cours de documentation, de nombreuses entités en dehors de ce contexte sont filtrées à tort ce qui entraîne une baisse du rappel à 0.85 (colonne **Dagenais R.**).

En comparaison, l'utilisation de notre version modifiée prenant en compte les thèmes présents dans le document permet de ne pas éliminer systématiquement les entités hors du package pour un rappel de 0.96 (colonne **thèmes R.**) et ce tout en maintenant une précision à 0.81 (colonne **thèmes R.**). Ce compromis rappel vs. précision semble alors plus acceptable.

8.6.4 Alignement des extraits de code

Comme nous l'avons montré dans le chapitre 2, les extraits de code Markdown sont souvent utilisés pour présenter des exemples d'utilisation. Ils contiennent donc des références aux entités de l'API en cours de documentation par l'écrivain.

Cet alignement permet de déterminer quelles sont les entités Nit auquel l'écrivain fait référence dans son extrait de code. Ceci sera utile pour suggérer des cartes en fonction du contenu de l'extrait de code. Il sera aussi utilisé pour trouver les entités Nit dont l'implémentation est similaire à l'extrait de code et ainsi suggérer le remplacement de l'extrait par une directive d'importation de code (section 7.3.2.4).

8.6.4.1 Alignement des extraits de code Markdown

Nous cherchons à aligner ces extraits de deux manières : 1) avec les entités Nit utilisées dans l'extrait de code Markdown ; et 2) avec les entités Nit dont le code source est le plus similaire avec celui de l'extrait de code Markdown.

Les extraits de code Markdown peuvent contenir des extraits compilables et d'autres non compilables. Nous utilisons deux processus différents pour aligner ces deux types d'exemples.

Analyse des extraits de code Nit compilables Les extraits de code comprenant du code Nit sont extraits depuis le Markdown puis passés à la chaîne de compilation suivant le même processus (présenté en section 8.3) que pour le corpus de code Nit

L'extrait de code est d'abord passé à l'analyseur syntaxique du langage afin d'obtenir un arbre syntaxique abstrait (AST). L'AST est ensuite enrichi par l'analyseur sémantique afin de lier les noeuds de l'arbre aux entités de l'API.

Une fois l'AST enrichi, il est utilisé pour exécuter une analyse statique du code source ayant pour but d'extraire les entités du modèle Nit utilisées dans le code d'exemple et d'identifier quelles utilisations sont faites de ces entités. Cette analyse est similaire à celle utilisée pour l'indexation du code des entités Nit (section 8.4.3).

Analyse des extraits de code non compilables Les extraits de code Markdown contenant du code Nit qui n'est pas compilable sont analysés grâce à un second *island parser* — le premier est utilisé pour extraire les identifiants dans le texte (section 8.6.3.1). Là aussi nous avons construit cet *island parser* grâce à un ensemble d'expressions régulières dédiées à la syntaxe du langage Nit.

L'analyse des extraits de code non compilables permet de retourner moins d'entités que par l'utilisation de la chaîne de compilation Nit. En effet, Nit utilisant de l'inférence de types, les types statiques sont peu présents dans le code, et donc l'*island parser* n'est pas en mesure d'associer les variables, paramètres et valeurs de retour aux types réellement utilisés.

Alignement avec les entités Nit Cet alignement permet de déterminer quelles sont les entités Nit utilisées dans l'extrait de code. Le vecteur de requête est composé à partir de chacune des entités référencées lors de l'analyse de son contenu en

utilisant les descripteurs d'action `name` et `full_name`. Les entités sont sélectionnées grâce à leur similarité avec le vecteur de requête dans l'index Nit.

Que l'extrait de code soit compilable ou non, nous limitons le bruit produit par l'utilisation des entités communément utilisées issues de la bibliothèque standard du langage, telles les entiers ou les chaînes de caractères, en appliquant un filtre basé sur le contexte du fichier README pour éliminer les conflits de noms. Ainsi, les entités en conflit qui ne sont pas en lien avec le package en cours de documentation par l'écrivain sont systématiquement supprimées.

Alignement avec le code source des entités Nit Cet alignement permet de déterminer quelles sont les entités Nit dont le code source d'origine est le plus similaire au code source de l'extrait de code Markdown.

Pour les extraits sur lesquels on peut exécuter l'analyseur syntaxique Nit, le vecteur représentant son code est récupéré grâce aux descripteurs d'action (section 8.4.3). Ce vecteur est ensuite comparé avec l'index afin de récupérer les entités les plus similaires au niveau de leur implémentation.

8.6.4.2 Validation de l'alignement des extraits de code

Validation de l'alignement avec les entités Nit Le tableau 8.7 présente le rappel et la précision de notre algorithme pour chaque package du corpus — nous ne présentons ici que les fichiers README contenant des extraits de code Nit. Les colonnes **R. base** et **P. base** indiquent respectivement le rappel et la précision de l'algorithme d'alignement des extraits de code compilables sans filtrage. Le rappel moyen est de 0.78, Cependant, une grande quantité de bruit est produite par l'algorithme de sélection conduisant à une précision de seulement 0.75. L'utilisation d'un *island parser* permet d'améliorer le rappel (colonne **R. I.P.**) à 0.95 au

Tableau 8.7: Performances de l'algorithme de sélection des entités de l'API depuis les extraits de code.

Légende: **R.** et **P.** représentent respectivement le rappel et la précision de l'algorithme ; **base** : algorithme sans filtrage ; **I.P.** : algorithme sans filtrage avec l'utilisation de l'*Island Parser* ; **standard** : algorithme avec *Island Parser* et filtrage sur la bibliothèque standard de Nit ; **contexte** : algorithme avec *Island Parser* et filtrage sur le contexte pour les conflits de noms ; **tous** : algorithme avec *Island Parser* et un filtrage sur la bibliothèque standard et sur le contexte.

	base		**I.P.**		**standard**		**contexte**		**tous**	
Bib.	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**
`android`	0.00	1.00	1.00	0.25	1.00	0.25	1.00	1.00	1.00	1.00
`app`	0.50	0.89	0.87	0.89	0.87	0.96	0.63	0.88	0.83	0.96
`github`	1.00	0.67	1.00	0.67	1.00	0.96	1.00	0.67	1.00	1.00
`json`	1.00	0.39	1.00	0.39	0.87	0.47	1.00	0.39	0.87	0.77
`nitcorn`	1.00	0.91	1.00	0.91	0.95	1.00	1.00	0.95	0.95	1.00
`nlp`	1.00	0.86	1.00	0.86	1.00	1.00	1.00	0.86	1.00	1.00
`popcorn`	0.68	0.94	0.84	0.79	0.83	0.84	0.84	0.81	0.83	0.86
`serial.`	0.86	0.42	0.86	0.28	0.81	0.32	0.86	0.58	0.81	0.58
`vsm`	1.00	0.70	1.00	0.70	0.85	1.00	1.00	0.70	0.85	1.00
Moyenne	**0.78**	**0.75**	**0.95**	**0.64**	**0.91**	**0.76**	**0.93**	**0.76**	**0.90**	**0.91**

détriment d'ajout de bruit puisque la précision descend à 0.64. L'ajout du filtre sur les entités de la bibliothèque Standard Nit et sur le contexte du document README permet de limiter le bruit produit (**P. tous** = 0.91), mais engendre toutefois la perte de résultats pertinents (**R. tous** = 0.90).

Tableau 8.8: Performances de l'algorithme de sélection des exemples de l'API depuis les extraits de code.

Bibliothèque	Rappel	Précision
`app`	1.00	1.00
`nitcorn`	1.00	1.00
`popcorn`	0.84	0.84
`serialization`	1.00	1.00
Moyenne	**0.96**	**0.96**

Validation de l'alignement avec le code source Notre corpus Nit contient seulement 31 extraits de code répartis en quatre packages qu'il est possible de remplacer par des exemples ou par le code des entités du modèle Nit. Le tableau 8.8 présente le rappel et la précision de notre algorithme pour ces packages, lesquels sont tous deux de 0.96. Tous les extraits de code qui ne sont pas alignés correctement sont dans le package `popcorn`.

Une revue manuelle des cas en erreur montrent que tous les extraits de code mal alignés sont des extraits contenant une seule ligne de code, donc trop peu d'information pour que le processus d'alignement fonctionne correctement. Cependant, tous les exemples de plus d'une ligne sont alignés comme ils le devraient. Nous croyons que ces résultats sont suffisants pour permettre la suggestion de substitution des extraits de code Markdown par une inclusion du code Nit pour les extraits qui en valent la peine, c'est-à-dire des extraits de plus de quelques lignes.

8.6.5 Alignement de la langue naturelle

L'alignement de la langue naturelle permet d'aligner le contenu des paragraphes, en-têtes et éléments de listes avec les entités de l'API. L'utilisation de la langue naturelle vient en complément de l'alignement des noms trouvés dans le texte et permet d'aligner les blocs Markdown qui ne contiennent aucune référence directe à une entité Nit (mise en relief, nom ou extrait de code).

Au même titre que l'alignement des mises en relief de code ou des identifiants, l'alignement de la langue naturelle permet de trouver quelles sont les entités Nit dont traite l'écrivain afin de lui suggérer les cartes de documentation d'API. Il permet en outre de trouver les entités Nit dont le commentaire est similaire au paragraphe en cours de rédaction afin de suggérer les cartes d'inclusion de commentaires (section 7.3.2.2) comme nous l'avons fait avec les extraits de code.

8.6.5.1 Alignement du texte du README

Pour chaque bloc de texte présent dans le fichier README, un vecteur représentant son contenu en langue naturelle est créé. On obtient ainsi la liste des entités les plus pertinentes par rapport au texte du bloc grâce à l'index Nit. Cette liste est triée selon la similarité $cos\,\theta$ de chacun des commentaires avec le bloc courant. Les entités ainsi obtenues sont ensuite filtrées afin de ne conserver que les résultats les plus pertinents.

Le premier filtre appliqué sélectionne les entités au dessus d'un seuil de pertinence calculé par la moyenne des similarités retournées plus l'écart type de ces similarités. On ne conserve alors que les entités dont le commentaire présente une forte similarité avec le texte en langue naturelle trouvée dans le fichier README.

Le second filtre est identique à celui utilisé pour l'alignement des identifiants trouvés dans le code (section 8.6.3) et dépend à la fois du contexte du bloc en cours d'alignement et des thèmes qui lui sont associés (section 8.6.1).

8.6.5.2 Validation de l'alignement de la langue naturelle

Nous avons annoté manuellement le corpus de fichiers README afin d'associer les blocs Markdown aux entités de l'API qu'ils sont supposés référencer et dont le commentaire est similaire. Le corpus une fois annoté permet de comparer les entités détectées par notre algorithme par rapport aux entités attendues par l'écrivain. Le tableau 8.9 présente le rappel et la précision de l'algorithme d'alignement de la langue naturelle pour chaque package du corpus.

Les colonnes **R. base** et **P. base** indiquent respectivement le rappel et la précision de l'algorithme de base sans aucune option de filtre. Bien que le rappel soit de 0.94, la précision de 0.05 est extrêmement faible. L'alignement avec les commentaires, sans filtre est inutilisable.

Le filtre sur la similarité minimale permet d'éliminer une grande partie du bruit (**seuil P.** = 0.43) mais engendre une diminution du rappel (**seuil R.** = 0.84).

Le filtre sur le contexte tel que proposé par Dagenais et Robillard (2012) permet d'atteindre une précision de 0.86 (colonne **Dagenais P.**) mais diminue encore un peu plus le rappel à 0.73 (colonne **Dagenais R.**). Là encore, c'est l'utilisation du contexte global au document qui filtre les entités externes au package en cours de documentation.

Nous avons aussi réutilisé l'algorithme modifié pour prendre en compte les thèmes du document tel que présenté pour l'alignement des noms (section 8.6.3) pour obtenir un rappel de 0.87 et une précision de 0.81. L'utilisation des thèmes permet d'atteindre un compromis rappel vs. précision que nous considérons plus acceptable.

Tableau 8.9: Performances de l'algorithme d'alignement de la langue naturelle.

Légende: **R.** et **P.** représentent respectivement le rappel et la précision de l'algorithme ; **base** : alignement sur le vecteur de commentaire ; **seuil** : filtrage sur le seuil de similarité ; **Dagenais** : filtrage sur le contexte tel que présenté dans l'article de Dagenais et Robillard (2012) ; **thème** : filtrage sur le contexte modifié pour prendre en compte les thèmes.

	base		**seuil**		**Dagenais**		**thèmes**	
Bibliothèque	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**	**R.**	**P.**
actors	0.98	0.03	0.86	0.67	0.73	0.71	0.94	0.89
ai	1.00	0.01	0.87	0.50	0.83	1.00	1.00	0.93
android	1.00	0.01	0.81	0.04	0.75	0.69	0.88	0.84
app	0.95	0.03	0.78	0.46	0.77	0.84	0.85	0.73
gamnit	0.93	0.00	0.72	0.19	0.66	0.81	0.96	0.84
geometry	0.94	0.05	0.85	0.61	0.72	0.77	0.97	0.81
github	0.97	0.04	0.89	0.26	0.70	0.85	0.78	0.91
ios	0.85	0.01	0.78	0.22	0.73	0.80	0.90	0.71
json	0.90	0.01	0.87	0.25	0.63	0.82	0.75	0.74
markdown	1.00	0.01	0.90	0.54	0.90	1.00	0.93	0.73
nitcorn	0.92	0.02	0.83	0.55	0.80	0.91	0.90	0.79
nlp	1.00	0.13	0.94	0.82	0.79	1.00	0.98	0.83
popcorn	0.81	0.16	0.74	0.24	0.52	0.63	0.78	0.80
posix	1.00	0.15	0.94	0.54	0.67	0.67	0.77	0.93
pthreads	0.81	0.03	0.74	0.53	0.66	1.00	0.75	0.78
sdl2	1.00	0.07	0.93	0.43	0.83	1.00	0.73	0.73
serialization	0.96	0.01	0.81	0.25	0.72	0.99	0.83	0.80
vsm	0.97	0.08	0.85	0.67	0.72	0.98	0.91	0.78
Moyenne	**0.94**	**0.05**	**0.84**	**0.43**	**0.73**	**0.86**	**0.87**	**0.81**

8.6.6 Consolidation de l'alignement

Chaque algorithme d'alignement présenté dans ce chapitre permet d'aligner certains blocs Markdown trouvés dans les fichiers README mais pas d'autres. En effet, des fichiers README contiennent des mises en relief de code d'autres non, certains fichiers utilisent des exemples de code ou pas, et ainsi de suite.

Afin d'obtenir les meilleurs résultats et ainsi permettre d'aligner un maximum de blocs dans chaque fichier, nous avons combiné les résultats de chacun des algorithmes d'alignement présentés.

Le cumul des résultats de chaque algorithme peut engendrer des conflits pour certains blocs. Par exemple, il se peut que les identifiants trouvés dans le texte ne correspondent pas aux entités alignées grâce à la langue naturelle extraite des commentaires. Afin de limiter de tels conflits, nous avons choisi de donner une plus grande importance aux résultats les plus fiables. Cette confiance est accordée dans l'ordre aux entités suivantes :

1. Entités alignées grâce aux mises en relief de code ;
2. Entités alignées grâce aux extraits de code ;
3. Entités alignées grâce aux identifiants ;
4. Entités alignées grâce à la langue naturelle.

Le tableau 8.10 présente le pourcentage de blocs identifiés par rapport aux entités de l'API grâce aux approches d'alignement proposées dans ce chapitre. La colonne **% id.** représente le nombre de blocs correctement alignés avec les entités de l'API, c'est-à-dire que toutes les entités attendues sont identifiées. La colonne **% partiels** représente le nombre de blocs partiellement alignés avec les entités de l'API, c'est-à-dire qu'au moins une des entités attendues est identifiée. La colonne **% mal id.** représente le nombre de blocs mal alignés avec les entités

Tableau 8.10: Pourcentage moyen de blocs Markdown identifiés grâce aux approches d'alignement avec les entités de l'API.

Légende: **% id.** : pourcentage de blocs correctement identifiés ; **% partiels** : pourcentage de blocs partiellement identifiés ; **% mal id.** : pourcentage de blocs mal identifiés. **% non id.** : pourcentage de blocs non identifiés.

Approche	id.	partiels	mal id.	non id.
Extraits code	4.0	7.7	13.9	74.4
Reliefs code	21.5	2.2	11.9	64.4
Identifiants	42.5	15.5	16.7	25.4
Langue naturelle	63.0	8.3	13.1	15.7
Tous	**69.0**	**15.9**	**1.8**	**13.3**

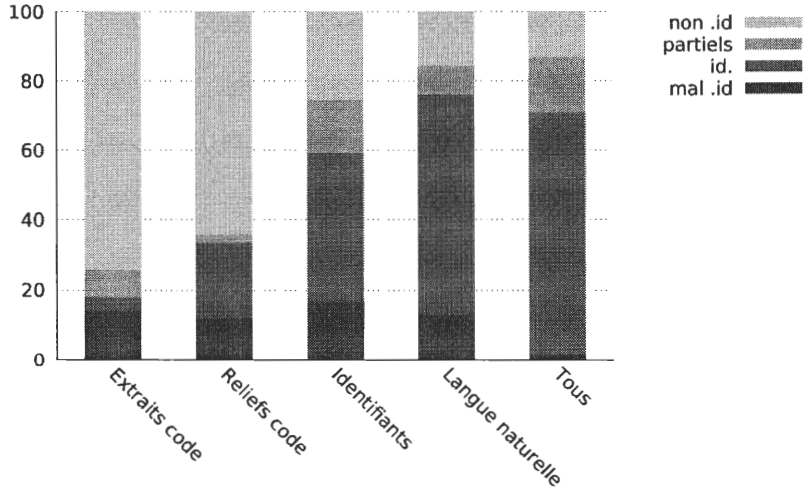


Figure 8.17: Pourcentage moyen de blocs Markdown identifiés grâce aux approches d'alignement avec les entités de l'API.

Légende: **% id.** : pourcentage de blocs correctement identifiés ; **% partiels** : pourcentage de blocs partiellement identifiés ; **% mal id.** : pourcentage de blocs mal identifiés. **% non id.** : pourcentage de blocs non identifiés.

de l'API, c'est-à-dire qu'aucune des entités retournées par les algorithmes n'est correcte par rapport aux entités attendues par l'écrivain.

La figure 8.17 compare le nombre de blocs Markdown alignés par chaque approche puis avec toutes les approches utilisées conjointement. Comme on peut le remarquer, l'utilisation de tous les algorithmes d'alignement combinés permet non seulement d'augmenter le maximum de blocs correctement identifiés et partiellement identifiés (total de 84.9%), mais aussi de limiter le nombre d'erreurs puisque peu de blocs sont alors mal identifiés (1.8%).

Ainsi, non seulement les entités identifiées par les approches d'alignement peuvent être combinées, mais cette combinaison permet d'obtenir de meilleurs résultats que chacune des approches utilisées indépendamment les unes des autres.

8.7 Menaces à la validité de notre étude

Validité de construction Nous avons repris la même méthodologie de validation que Dagenais et Robillard (2012) et Rigby et Robillard (2013) pour la création et l'annotation du corpus. Nous croyons donc que la menace à la validité de construction de notre étude est minime.

Validité interne La menace majeure à la validité interne de notre étude est le processus d'annotation du corpus de fichiers README Nit. En effet, le corpus est annoté, et nous n'en avons utilisé qu'un seul. Le manque de recouvrement entre les annotations placées peut aussi constituer un biais dans nos résultats : il est en effet possible qu'un autre annotateur produise des regroupements différents. Notons que ni l'étude de Dagenais et Robillard (2012), ni l'étude de Rigby et Robillard (2013) ne précisent le nombre d'annotateurs.

Validité externe Notre échantillon d'analyse de 18 projets est trop restreint pour nous permettre de généraliser l'efficacité de notre approche à tous les README. C'est malheureusement l'ensemble des fichiers README existant pour des projets Nit et pour lesquels nous disposons du code source.

Cependant, nous avons comparé le contenu de notre corpus avec celui provenant de notre étude empirique des fichiers README trouvés sur GitHub (chap. 2). Nos métriques montrent une similarité entre nos projets et les projets ayant entre 10 et 99 étoiles sur GitHub. On peut alors espérer des résultats identiques pour ces fichiers puisqu'ils contiennent le même type d'information et dans les mêmes proportions.

En revanche, certaines pratiques, notamment l'utilisation abondante des mises en relief de code, peuvent être spécifiques aux us et coutumes du langage Nit. Nous ne pouvons pas non plus généraliser notre approche à des README écrits pour des projets dans d'autres langages de programmation.

8.8 Conclusion

Dans ce chapitre, nous avons présenté notre approche d'alignement du contenu des fichiers README avec les entités de l'API Nit.

Le contenu du modèle Nit est tout d'abord indexé lors de la phase *hors ligne* afin d'accélérer l'alignement du contenu Markdown en cours d'écriture par l'écrivain lors de la phase *en ligne*. L'index ainsi créé permet d'accéder aux entités du code Nit grâce à leur nom (court, canonique ou lemmatisé) ou un vecteur représentant les lemmes utilisés dans les commentaires ou les constructions utilisées dans le code.

La phase en ligne utilise l'index pour *aligner* le contenu Markdown du fichier README avec les entités du modèle Nit. Cet alignement se base sur les étapes

d'analyse des thèmes du README, des mises en relief de code, des identifiants trouvés dans le texte, de la langue naturelle et des extraits de code utilisés.

Pour chaque étape, nous avons analysé le rappel et la précision des algorithmes utilisés grâce à un corpus de fichiers README issus du projet Nit annoté manuellement. Nous avons montré la corrélation de notre corpus avec celui issu de GitHub présenté au chapitre 2 afin d'assurer qu'il soit représentatif de fichiers README de la vie réelle.

L'utilisation des thèmes pour le filtrage des entités permet de limiter les faux positifs obtenus lors de l'alignement des identifiants et de la langue naturelle. Cette approche est plus efficace que celle proposée par Dagenais dans notre contexte.

Enfin, nous avons présenté la dernière étape du processus, l'alignement du document complet, qui utilise le résultats des étapes précédentes afin de propager et filtrer les liens vers les entités du modèle selon leur degré de confiance. Nous avons montré que notre approche permet d'identifier 84.9% des blocs contenus dans les fichiers README avec seulement 1.8% d'erreur.

Dans le prochain chapitre, nous présentons le processus de génération des cartes de suggestions, dont l'interface a été présentée au chapitre 7, processus qui s'appuie sur le processus d'alignement et l'index décrits dans le présent chapitre.

CHAPITRE IX

`DOC_SUGGEST` : UNE APPROCHE POUR SUGGÉRER LES CARTES DE DOCUMENTATION DES FICHIERS README

Dans le chapitre 7, nous avons présenté le manuel d'utilisation des outils de génération et de maintenance de fichiers README : `nitreadme` permet de générer des fichiers README à partir de zéro puis d'améliorer les fichiers existants en insérant automatiquement les suggestions dans le texte ; `nitweb/readme` offre un éditeur de fichiers README interactif affichant des cartes de suggestions de documentation au cours de la rédaction. Le chapitre 8 expliquait l'approche utilisée par la bibliothèque `doc_align` pour créer un alignement entre les entités du modèle Nit et le contenu textuel du fichier README. Dans le présent chapitre, nous expliquons comment le module `doc_suggest` utilise cet alignement pour suggérer des cartes de documentation à l'écrivain et les insérer dans le texte, aussi bien en mode interactif qu'en ligne de commande. La figure 9.1 illustre la position de `doc_suggest` dans l'ensemble de l'écosystème Nit.

Les difficultés liées à l'écriture de la documentation décrites au chapitre 1 montrent que l'écrivain a parfois du mal à trouver quoi expliquer dans sa documentation. Le premier objectif des cartes de documentation est donc d'aider à trouver le contenu qu'il est pertinent de documenter. Le second objectif de ces cartes est de maintenir le contenu du fichier README synchronisé avec le code. Notre approche vise donc soit à proposer du contenu à l'écrivain que les outils sont capables de maintenir à

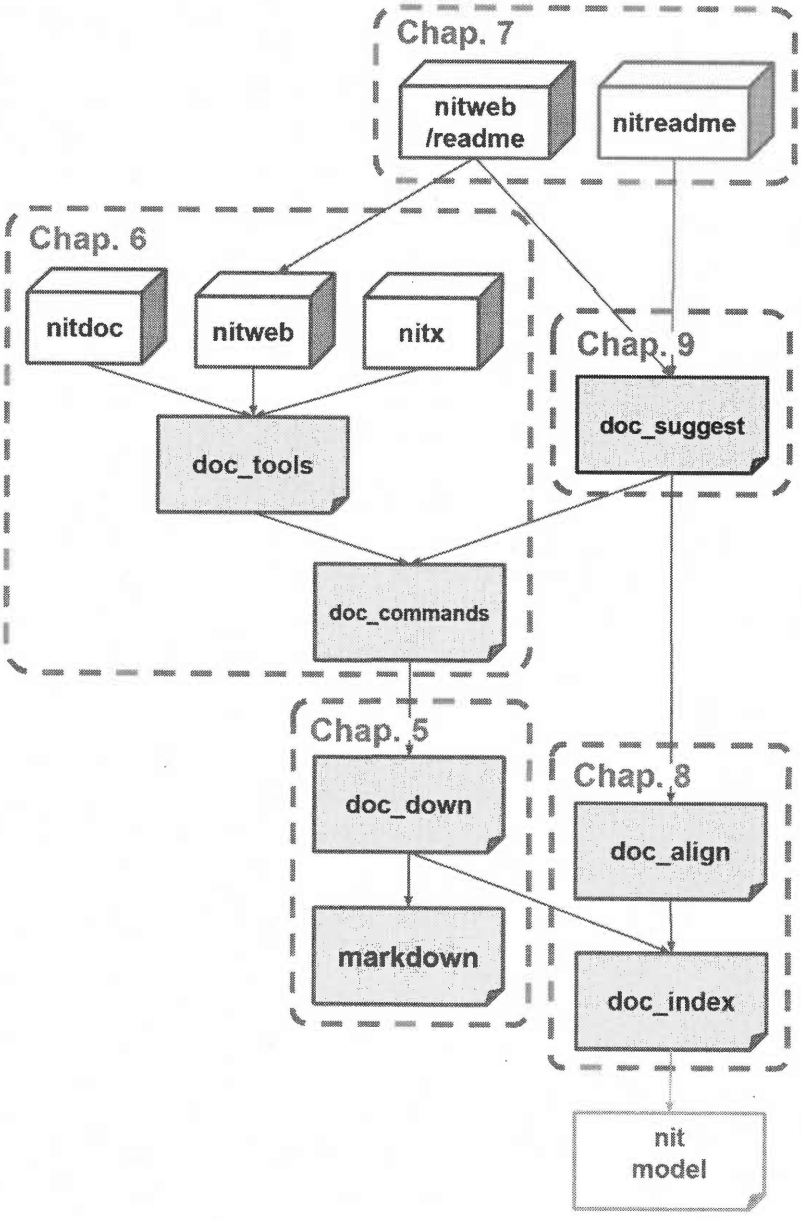


Figure 9.1: Place du module `doc_suggest` dans l'écosystème de documentation Nit.

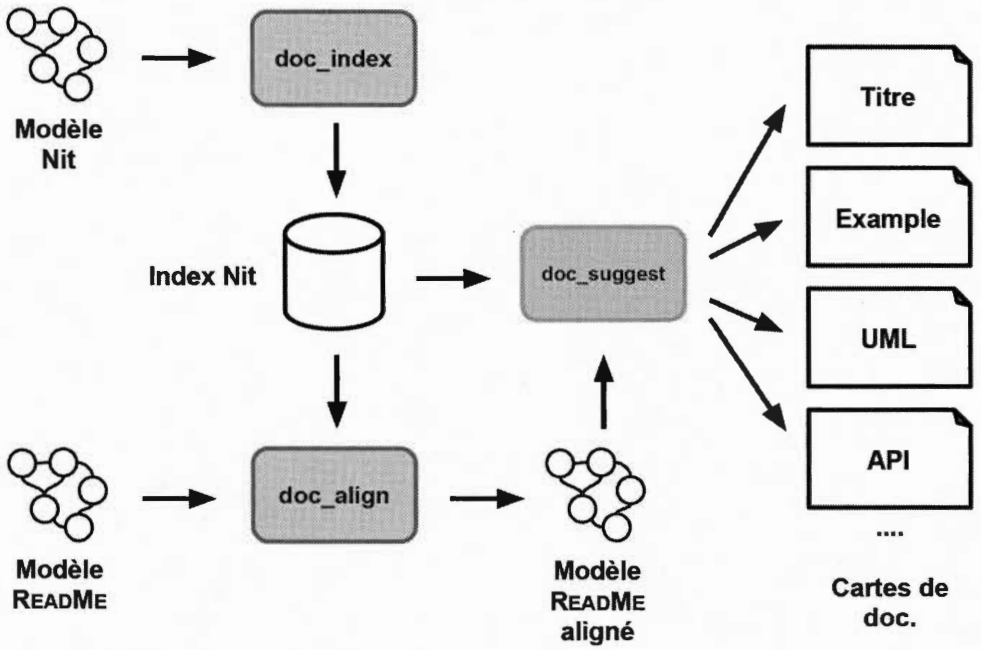


Figure 9.2: Processus de suggestion des cartes de documentation.

jour, soit à remplacer du contenu existant par du contenu plus facile à maintenir à jour.

La figure 9.2 illustre la position de `doc_suggest` par rapport aux autres étapes pour la production des suggestions de cartes de documentation. La création de ces suggestions se base sur le contenu du modèle Nit qui est accédé grâce à l'index créé par `doc_index` durant l'*étape hors ligne* (section 8.4). Ces suggestions sont sélectionnées et insérées en fonction de l'alignement effectué par `doc_align` (section 8.6).

Les suggestions sont basées sur le bloc Markdown en cours de rédaction par l'écrivain. Ce bloc est identifié grâce à la position du curseur dans l'interface `nitweb` pour le mode interactif. À partir de ce bloc, `doc_suggest` détermine quelles suggestions afficher à l'écrivain.

La section 9.1 dresse l'état de l'art concernant la suggestion. Le processus de suggestion des cartes de documentation pour la version interactive de l'outil est présenté à la section 9.2. La version modifiée de ce processus pour la génération *a priori* de nouveaux fichiers README est expliquée à la section 9.3, tandis que la version modifiée pour l'amélioration *a posteriori* des fichiers existants est expliquée à la section 9.4. Enfin, la section 9.5 présente certaines validations que nous avons effectuées et les résultats obtenus.

9.1 État de l'art de la suggestion de documentation

Sadoun *et al.* (2016) proposent de générer automatiquement les sections principales du README pour des analyseurs de la langue naturelle en déduisant des ontologies depuis un ensemble de fichiers README existants pour ce type d'outils. Leur approche est spécifique au cas des analyseurs de la langue naturelle et ne propose aucune solution pour la maintenance.

Treude et Robillard (2016) proposent d'améliorer la documentation d'API en y incluant des phrases perspicaces (*insight sentences*) minées depuis le site d'échange StackOverflow. Les phrases sont choisies grâce à l'apprentissage machine à partir d'un jeu d'entraînement construit manuellement. Elles sont ajoutées dans la documentation d'API en dessous du commentaire de l'entité lors de sa génération par le lecteur. Toutefois, les approches basées sur le contenu de StackOverflow — en fait, *crowdsourcées* en général — sont inefficaces en ce qui concerne la documentation d'un nouveau logiciel pour lequel il n'existe pas encore de base d'utilisateurs.

Dagenais et Robillard (2012) proposent RecoDoc, une extension au navigateur web permettant d'ajouter automatiquement des liens vers la documentation d'API dans le texte que parcourt le lecteur. Nous avons comparé nos techniques d'alignement du contenu de la documentation avec les leurs au chapitre précédent (chap. 8).

Long *et al.* (2009) et Zhang *et al.* (2011) comparent la structure des API pour suggérer des liens vers les API similaires. Cette technique est utilisée pour ajouter des listes de liens de type *related also*. Encore une fois, la recommandation se fait pour le lecteur et non pour l'écrivain.

Kim *et al.* (2013) proposent de miner les bases de code à la recherche d'exemples et de tests unitaires à inclure dans la documentation d'API. Leur outil, eXoaDocs, reprend les mêmes informations que Javadoc en y ajoutant des exemples d'utilisation tirés d'un corpus. Cependant, mise à part l'inclusion dans la page de documentation, rien n'est fait pour aider l'écrivain à personnaliser ou maintenir son exemple.

Eriksson *et al.* (2002) proposent un navigateur spécifique pour la documentation d'API. Pour chaque entité visualisée dans le navigateur, l'outil sélectionne les informations importantes depuis la documentation d'API et les affiche dans différents formats de présentation. Cette vue propose une abstraction du code qu'elle documente et permet au lecteur de mieux se représenter une API complexe.

Les études se concentrent généralement sur une approche *a posteriori* où la documentation existe déjà — avec des défauts — et l'outil tente de corriger ces défauts. Robillard *et al.* (2017) affirment le besoin d'une documentation à la demande par le lecteur pour lutter contre le manque de documentation existante. Encore une fois l'effort est mis sur une solution *a posteriori*. Nous croyons cependant qu'une solution *a priori* d'assistance à l'écrivain peut limiter le besoin pour une solution corrective. Notons toutefois que la solution que nous proposons n'exclut pas l'utilisation d'une approche *a posteriori* comme nous l'avons montré avec l'assistant de lecture `nitweb/readit` (section 7.2.2).

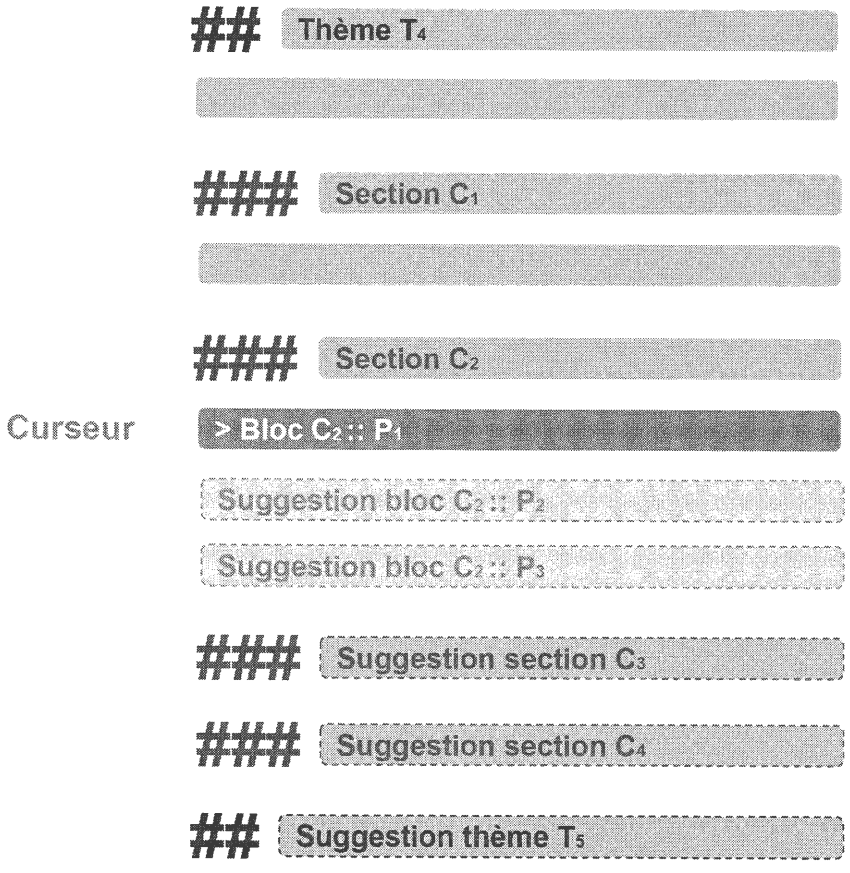


Figure 9.3: Illustration des blocs considérés pour la génération des suggestions des cartes de documentation.

9.2 Processus de suggestion interactif

À chaque modification du fichier dans l'interface, `nitreadme` analyse le contenu Markdown puis obtient l'alignement du texte grâce à `doc_align`. On sait alors à quelles entités Nit font références les éléments du Markdown et quels sont les thèmes des différentes sections.

La figure 9.3 donne un exemple de document en cours de rédaction par l'écrivain. Son curseur est actuellement dans le bloc C_2`::`P_1 (en orange dans la figure). Il

documente donc la propriété P_1 de la classe C_2. À partir de ce bloc on peut déterminer les cartes pertinentes à suggérer à l'écrivain.

9.2.1 Suggérer les cartes d'échafaudage

La suggestion des cartes d'échafaudage se base sur ce que l'écrivain a déjà rédigé dans sa documentation afin de lui proposer de nouvelles sections qu'il faudrait ajouter au document. Suggérer une carte d'échafaudage à l'écrivain revient alors à déterminer quelles sections il a déjà rédigées et quelles sections restent à rédiger.

Nous établissons le type et l'ordre des sections d'échafaudage selon notre étude empirique des fichiers README issus de GitHub (chap. 2). Pour chaque section d'échafaudage, nous listons les suggestions de cartes de documentations associées et sa section correspondante dans le manuel d'utilisation de `nitreadme` présenté au chapitre 7 :

1. **Titre** : *carte de titre* (section 7.3.1.1) ;
2. **Sommaire** : *carte de sommaire* (7.3.1.2) ;
3. **Introduction** : combinaison de *cartes de documentation d'API* (section 7.3.2) détaillée à la section 9.2.2 ;
4. **Exemples d'utilisation** : combinaison de *cartes d'inclusion de code* (7.3.2.4) détaillée à la section 9.2.2 ;
5. **Démarrage** : *carte de démarrage* (7.3.1.3) ;
6. **API et fonctionnalités** : combinaison de *cartes de documentation d'API* (section 7.3.2) détaillée à la section 9.2.2 ;
7. **Problèmes** : *carte de problèmes* (7.3.1.4) ;
8. **Auteurs** : *carte d'auteurs* (7.3.1.5) ;
9. **Contribuer** : *carte de contribution* (7.3.1.6) ;
10. **Tests** : *carte de tests* (7.3.1.7) ;
11. **Licence** : *carte de licence* (7.3.1.8).

La section en cours de rédaction par l'écrivain est déterminée grâce aux thèmes alignés, tel que décrit dans le chapitre précédent (section 8.6.1). La section de niveau 2 (en violet dans la figure 9.3) à laquelle la section en cours de rédaction appartient contient majoritairement le thème T_4 (ex : *API et fonctionnalités*). L'outil considère alors que l'écrivain est en train de rédiger la documentation relative à ce thème.

À partir de cette section, on propose la suggestion vers la section suivante si on possède les informations nécessaires pour produire le contenu de la carte. Dans notre exemple de la figure 9.3, on suggère donc la section de niveau 2 suivante relative au thème T_5 (ex : *Problèmes*).

Si les informations pour générer une carte d'échafaudage ne sont pas disponibles, alors on sélectionne la section suivante dans l'ordre et ainsi de suite jusqu'à la fin

du document. De même si l'écrivain choisit de rejeter une suggestion en cliquant sur le bouton *Rejeter* (section 7.3).

9.2.2 Suggérer les cartes de documentation d'API

Les sections *Introduction*, *Exemples d'utilisation* et *API et fonctionnalités* sont composées de plusieurs suggestions de cartes de documentation d'API. Elles permettent à l'écrivain de documenter les fonctionnalités de son projet grâce au code source et aux entités Nit qu'il contient via l'inclusion d'extraits issus de l'auto-documentation d'API.

Dans notre exemple de la figure 9.3 (p. 261), en fonction des entités Nit référencées dans le texte du bloc courant $\texttt{C}_2\texttt{::P}_1$, de ses voisins et de ses parents ($\texttt{C}_2$), `nitreadme` suggère des cartes de documentation d'API pour les entités qu'il juge pertinentes pour les blocs suivants. Ici, après avoir traité de la propriété $\texttt{C}_2\texttt{::P}_1$ (en orange dans la figure), `nitreadme` suggère des cartes pour les propriétés $\texttt{C}_2\texttt{::P}_2$ et $\texttt{C}_2\texttt{::P}_3$.

`nitreadme` peut aussi proposer de nouvelles sections à ajouter au document basées sur la documentation d'entités liées à celles déjà présentes dans le code. Comme le fichier contient déjà de la documentation sur les classes $\texttt{C}_1$ et $\texttt{C}_2$, il propose des sections pour les classes $\texttt{C}_3$ et $\texttt{C}_4$ (en bleu dans la figure).

Le nombre et la nature des sections d'API utilisées dépend de la structure du projet à documenter. Comme le montre la figure 9.4, pour un projet contenant de nombreux groupes, on souhaite que le README donne un tour d'horizon des groupes disponibles et explique à quoi ils servent. Dans un projet avec peu de groupes mais de nombreux modules, on veut présenter les modules. Dans le cas d'un projet contenant un seul module, ce sont les classes qu'il est plus important de présenter. Enfin, si le projet ne contient qu'une seule classe, on se concentre alors sur ses propriétés. Les suggestions de cartes de documentation d'API prennent en

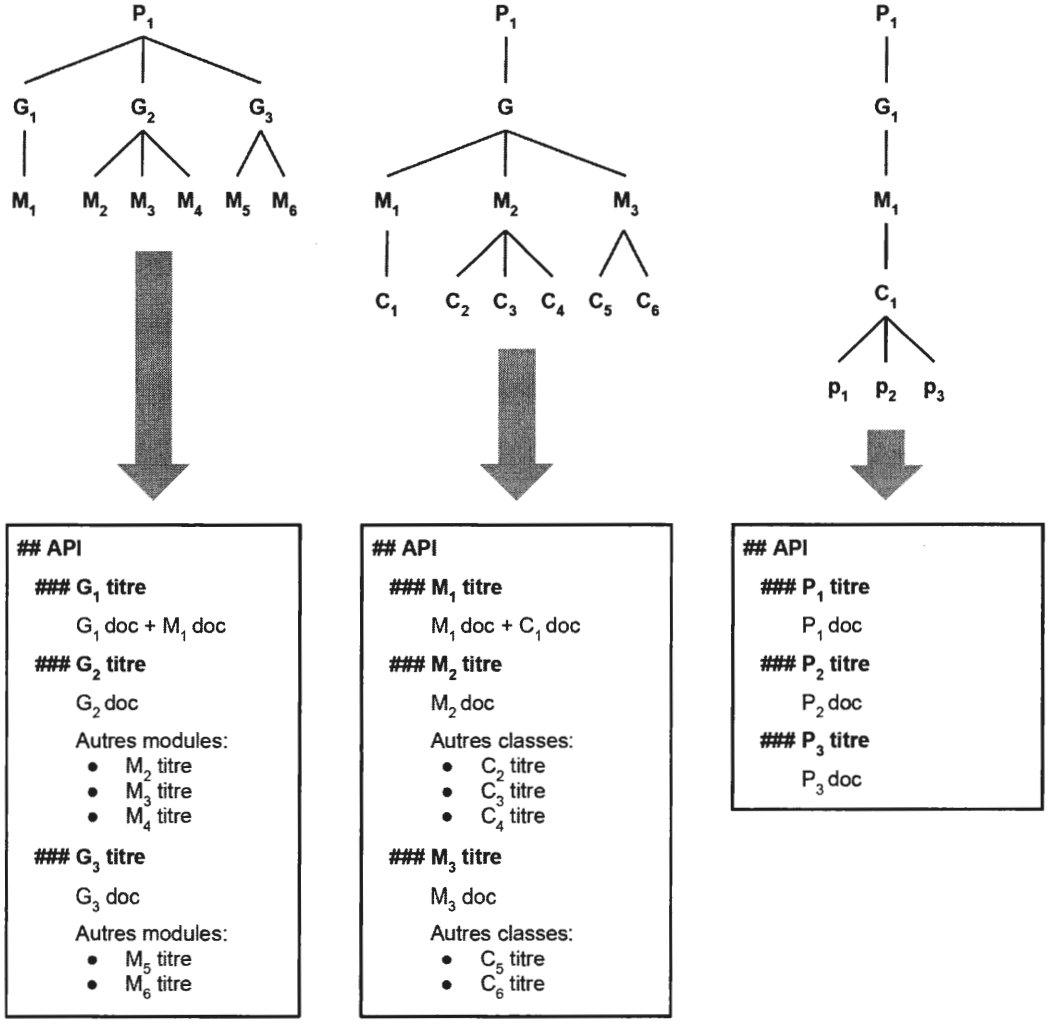


Figure 9.4: Heuristique de transformation de la structure du package en suite de sections Markdown pour la documentation de l'API.

compte cette structure afin de proposer les cartes les plus pertinentes à l'écrivain en fonction de ses besoins.

Le choix des cartes de documentation d'API se fait aussi en fonction de plusieurs thèmes ajoutés afin que l'écrivain puisse déclencher manuellement la suggestion d'une carte. Ces thèmes sont listés en fonction de la carte qu'ils déclenchent :

— Cartes de liens : `/link/`, `/nitdoc/`, `/nitweb/` ;
— Cartes de commentaires : `/comment/` ;
— Cartes de listes : `/list/` ;
— Cartes d'UML : `/diagram/`, `/uml/`, `/inherit/`, `/extend/`, `/import/` ;
— Cartes de code et d'exemples : `/example/`.

On collecte les suggestions de chaque types de cartes et on les affiche à l'écrivain, puis on attend qu'il insère une carte ou entre du nouveau contenu. Les cartes déjà ajoutées au document ne sont jamais proposées à nouveau. De même pour les cartes qui ont été rejetées par l'écrivain à l'aide du bouton *Dismiss*.

9.2.2.1 Liens vers la documentation d'API

Afin de ne pas surcharger le texte de la documentation nous choisissons d'encourager l'utilisation d'un seul lien par entité, lors de sa première apparition dans le texte. Ainsi, si un lien existe déjà pour une entité, aucune suggestion n'est faite à nouveau.

Par exemple, pour un lien vers la classe `MdParser` où l'option `text` est automatiquement remplacée par le mot à l'origine de la référence à l'entité Nit dans le texte Markdown : `[[markdown::MdParser | text: parser]]`

9.2.2.2 Inclusion de commentaire

Les cartes d'inclusion de commentaire visent à satisfaire deux objectifs : 1) suggérer à l'écrivain du contenu à ajouter dans sa documentation, et ce en lui affichant le texte

issu des commentaires afin de lui rappeler ce que contient l'API ; et 2) si le texte correspond déjà à ce que veut exprimer l'écrivain, lui permettre d'insérer la carte dans sa documentation et éviter ainsi de se répéter.

Suggestion de remplacement Chaque paragraphe est déjà aligné aux entités avec un commentaire similaire grâce au module `doc_align`. On propose alors des cartes pour les commentaires des entités alignées grâce à la langue naturelle avec une similarité supérieure à 0.75 afin de limiter le bruit.

Suggestion d'ajout Pour la suggestion de nouveau contenu, on considère les entités déjà référencées dans le texte ou utilisées dans une carte comme étant déjà documentées. On cherche alors de nouvelles entités à suggérer à l'écrivain. Pour cela, on s'appuie sur la structure du projet à documenter afin de déterminer si la carte à suggérer doit concerner un groupe, un module, une classe ou une propriété et on sélectionne les entités pourvues d'un commentaire. L'ordre des entités est déterminé en fonction de nos observations du chapitre 4 : ordre lexicographique pour les noms des groupes et modules, ordre du code pour les classes et propriétés.

9.2.2.3 Liste d'entités

L'objectif des listes est d'ajouter des références vers des entités similaires à celles identifiées dans le texte mais pour lesquelles il n'est pas pertinent de consacrer un paragraphe complet. La suggestion des listes propose ainsi d'ajouter des liens vers des entités en rapport avec celles dont traite l'écrivain mais qui ne sont pas déjà citées. Ces listes sont utilisées pour implémenter les *Voir aussi* (*See also*) que l'on trouve dans les outils d'auto-documentation d'API.

Pour ce faire, on considère les entités déjà référencées dans le texte ou utilisées dans une carte. On sélectionne alors les entités en rapport comme les super ou sous-classes, les méthodes de la même classe, etc.

9.2.2.4 Inclusion de code

La carte d'inclusion de code est utilisée pour suggérer des exemples d'utilisation à insérer dans le fichier README. Cette carte permet aussi à l'écrivain de remplacer un extrait de code Markdown par l'implémentation d'une entité — ou d'un exemple — dans le code source.

Suggestion de remplacement Chaque extrait est déjà aligné aux entités avec un code similaire grâce au module `doc_align`. On sélectionne alors l'exemple — ou toute autre entité — dont le code le plus similaire, mais dont la similarité cosinus est supérieure à 0.75 afin de limiter le bruit.

Suggestion d'ajout Pour la suggestion de nouveaux exemples, on regarde le contexte du bloc courant afin de sélectionner les entités dont traite l'écrivain. On obtient ensuite la liste des exemples pertinents pour ces entités depuis l'index Nit. Dans le cas de la section *Exemples d'utilisation*, le contexte utilisé est celui du package en cours de documentation.

Pour chaque exemple trouvé, s'il est déjà utilisé par une autre carte dans le fichier ou si un extrait de code Markdown du fichier est déjà aligné avec le code de l'exemple, alors il est ignoré afin de ne pas le proposer deux fois.

9.2.2.5 Diagrammes UML

Le type de diagramme est déterminé en fonction du type des entités Nit référencées dans le texte :

- une majorité de packages : diagramme de packages pour montrer les dépendances d'importation enfants/parents des packages cités ;
- une majorité de groupes : diagramme de groupes pour montrer le contenu du package en cours de documentation ;
- une majorité de modules : diagramme de groupes pour montrer le contenu du

package ou du groupe en cours de documentation ;

— une majorité de classes : diagramme de classes pour montrer la hiérarchie d'héritage des classes citées.

En fonction du type de diagramme, toutes les entités des contextes sont ajoutées jusqu'à l'atteinte du seuil — arbitrairement fixé à 15 afin de conserver des diagrammes lisibles. Par exemple, pour un diagramme de classes, toutes les classes des contextes locaux, voisins et parents sont ajoutées jusqu'à un maximum de 15 classes.

Si le maximum de 15 entités n'est pas atteint, on ajoute au diagramme d'autres entités du package en cours de documentation : d'abord les parents et enfants, par exemple, les super-classes d'une classe ou les modules important un module s'il y en a ; sinon les autres membres, comme les autres classes d'un module ou les autres modules d'un groupe.

La carte de documentation générée appelle la commande de création d'un diagramme UML en précisant le package en cours de documentation et les entités à dessiner. Les options de la carte peuvent alors être modifiées par l'écrivain afin d'ajouter ou de supprimer des entités du diagramme UML.

Par exemple, pour un diagramme de classes :

```
[[uml: markdown | entities: MdRenderer;HTMLRenderer;
   LatexRenderer;ManRenderer;MarkdownRenderer;RawRenderer]]
```

9.3 Processus *a priori* de génération d'un nouveau fichier README

La génération *a priori* d'un nouveau README consiste à choisir un package à documenter puis à lancer le processus de suggestion et à accepter — ou rejeter — automatiquement les suggestions, et ce jusqu'à compléter le document. On injecte ainsi l'ensemble des sections d'échafaudage pertinentes pour le package. Par exemple, on n'insère la section *Tests* que si le package contient des tests unitaires. En revanche, on affiche le conseil associé à l'écrivain dans la console.

La seule différence avec le processus de suggestion se trouve dans la génération de la section *API*. En effet, on ne veut pas forcément insérer l'ensemble de toutes les cartes de documentation d'API possibles pour le package — on souhaite seulement donner une vue au niveau du contenu du package. Ainsi, les heuristiques suivantes sont appliquées pour limiter le nombre de suggestions insérées :

1. Si le package contient plus d'un groupe, on insère la documentation des groupes contenant un commentaire, la liste des autres groupes, la liste des exemples pour chaque groupe et un diagramme UML représentant la hiérarchie d'importation des groupes ;
2. Sinon, si le package contient plus d'un module, on insère la documentation des modules contenant un commentaire, la liste des autres modules, la liste des exemples pour chaque module et un diagramme UML représentant la hiérarchie d'importation des modules ;
3. Sinon, si le package contient plus d'une classe, on insère la documentation des classes publiques contenant un commentaire, la liste des autres classes publiques, la liste des exemples pour chaque classe et un diagramme UML représentant la hiérarchie d'héritage des classes publiques ;
4. Enfin, si le package contient seulement des propriétés, on insère la documentation des propriétés publiques contenant un commentaire, la liste des autres propriétés publiques et un exemple d'utilisation de chaque propriété.

Ces heuristiques sont fondées sur notre analyse empirique des fichiers README issus de GitHub (chap. 2) ainsi que sur nos expérimentations sur la génération automatique de fichiers README (chap. 4).

9.4 Processus *a posteriori* d'amélioration d'un README existant

Le processus d'amélioration *a posteriori* consiste à insérer les suggestions pertinentes dans un fichier README existant. Pour cela, on avance dans le texte et on applique les

modifications en fonction de suggestions qui prennent en compte le contexte global au document. Ici, l'ordre des sections n'est pas strict ; on essaye simplement de composer avec celui déjà utilisé dans le document.

Le processus se déroule comme suit :

1. Détection des sections d'échafaudage déjà présentes grâce aux thèmes du README ;
2. Réorganisation des sections existantes selon l'ordre préconisé ;
3. Remplacement des informations existantes par des cartes de liens, commentaire, listes et exemples ;
4. Insertion des sections d'échafaudage manquantes sauf l'API ;
5. Insertion des cartes d'API pertinentes aux emplacements des références présentes dans le texte.

9.5 Validation

Dans cette section, nous expliquons la démarche suivie pour valider notre approche. Nous avons effectué une validation spécifique à chaque type de génération afin de valider aussi bien le processus de suggestion interactif que celui d'échafaudage *a priori* automatique que celui d'amélioration *a posteriori*.

Validation du mode de suggestion interactif La validation des suggestions interactives a été faite grâce à deux experts du langage Nit. Nous leur avons demandé de réécrire le fichier README d'une bibliothèque existante dont ils sont l'auteur.

La liste des bibliothèques et leur date de dernière modification par l'expert est la suivante :

— **Expert 1** : bibliothèque `pthreads` – 07/04/2015

— **Expert 2** : bibliothèque `ai` – 28/08/2015

Nous avons donné à chaque expert les sources de sa bibliothèque et l'outil `nitreadme` avec son interface de suggestion interactive, avec la consigne d'écrire un fichier README

pour la bibliothèque. Chaque expert disposait d'une heure et demie pour compléter son fichier.

Validation du mode de suggestion *a priori* Pour la validation du mode *a priori*, nous avons sélectionné aléatoirement 50 packages issus du projet Nit ne contenant aucun fichier README. Pour chaque package, nous avons généré le squelette du fichier à l'aide de `nitreadme` avec l'option `--scaffold` et ainsi obtenu 50 nouveaux fichiers README. Nous les avons soumis pour inclusion au dépôt Nit sous la forme de requêtes d'intégration, puis nous avons demandé aux concepteurs du langage d'accepter ou de rejeter les fichiers README proposés. La liste des projets est fournie en annexe F.1.

Validation du mode de suggestion *a posteriori* Nous avons réutilisé le corpus des 18 projets Nit contenant déjà un fichier README tel qu'il est présenté au chapitre 8. Pour chacun des projets, nous avons utilisé `nitreadme` avec l'option `--enhance` afin d'obtenir un fichier README amélioré. Nous avons ainsi obtenu 18 fichiers README modifiés que nous avons soumis pour ajout au dépôt Nit sous la forme de requêtes d'intégration, puis nous avons demandé aux concepteurs du langage d'accepter ou de rejeter les fichiers README proposés.

9.5.1 Contenu des fichiers générés

Le tableau 9.1 donne le nombre d'occurrence de chaque type de carte d'échafaudage utilisée pour générer ou améliorer les fichiers README.

Parmi toutes les sections d'échafaudage, nous avons désactivé la suggestion des sections *Problèmes*, *Contribuer* et *Licence* qui sont identiques à tous les packages dans le projet Nit et déjà documentées dans le fichier README principal du dépôt du langage. Pour les autres sections, on peut remarquer que les titres, introductions et auteurs sont présents dans tous les fichiers générés. Peu de projets contiennent des tests, l'outil suggère

Tableau 9.1: Occurrences de chaque section d'échafaudage insérée par les experts (colonne **experts**) ou générée par l'outil `nitreadme` (colonnes **--scaffold** et **--enhance**).

Carte Échafaudage	experts	--scaffold	--enhance
Titre & Intro	2	50	18
Sommaire	2	32	18
Exemples	2	18	18
Installation	1	5	6
Auteurs	2	50	18
Tests	0	6	4

donc rarement la section d'échafaudage sur le lancement des tests. Les procédures d'installation sont les moins fréquentes : la base de code contient plus de bibliothèques que de programmes.

Le tableau 9.2 donne le nombre d'occurrences de chaque type de carte de documentation d'API utilisée pour générer ou améliorer les fichiers README.

Les experts préfèrent les cartes d'inclusion de commentaires et de liens aux autres types de cartes ; les exemples sont peu utilisés. Ce constat peut être lié au fait que les packages Nit ne contiennent pas beaucoup d'exemples à l'heure actuelle et ce, même suite à l'adoption de notre spécification. Les diagrammes UML sont quant à eux les moins utilisés.

Pour la génération de nouveaux fichiers, aucun lien n'est suggéré, on se contente de ceux déjà présents dans les cartes de commentaires et de listes. Les cartes d'insertion de commentaires sont présentes dans tous les 50 documents afin de composer l'introduction. On compte en plus 38 sections relatives à l'API générée. Les listes sont utilisées au sein de la description d'API pour présenter des ensembles de classes et de propriétés. Tous les exemples utilisés le sont dans la section *Exemples d'utilisation* ; aucun n'est

Tableau 9.2: Occurrences de chaque carte de documentation d'API insérée par les experts (colonne **experts**) ou générée par l'outil `nitreadme` (colonnes **--scaffold** et **--enhance**).

Carte API	experts	--scaffold	--enhance
Lien	10	0	110
Commentaire	10	88	118
Liste	5	56	25
Exemples	3	18	30
UML	1	24	19
Total	**29**	**186**	**302**

présenté au sein de l'API. Enfin, un peu moins de la moitié des fichiers générés contient un diagramme UML. Ceux-ci sont réservés pour les packages les plus gros.

9.5.2 Comparaison des fichiers générés

Nombre de lignes La figure 9.5 compare le nombre de lignes moyen des fichiers README produits par les experts et compilés vers du Markdown brut avec ceux du corpus Nit utilisé au chapitre 8 et ceux issus de GitHub au chapitre 2.

On peut remarquer que les fichiers produits à l'aide de l'interface interactive de `nitreadme` contiennent autant de lignes que les projets de plus de 10 000 étoiles et ceux de la liste *awesome*, donc beaucoup plus que les fichiers produits manuellement.

Avec une moyenne de 112 lignes par README, les fichiers générés par `nitreadme --scaffold` sont comparables aux fichiers Nit écrits manuellement (136 lignes) et entre ceux des projets GitHub comprenant entre 10 et 999 étoiles (80 à 141 lignes). Les fichiers améliorés par `nitreadme --enhance` sont quand à eux beaucoup plus gros avec 433 lignes, donc bien plus que les projets de la liste *awesome* (245 lignes).

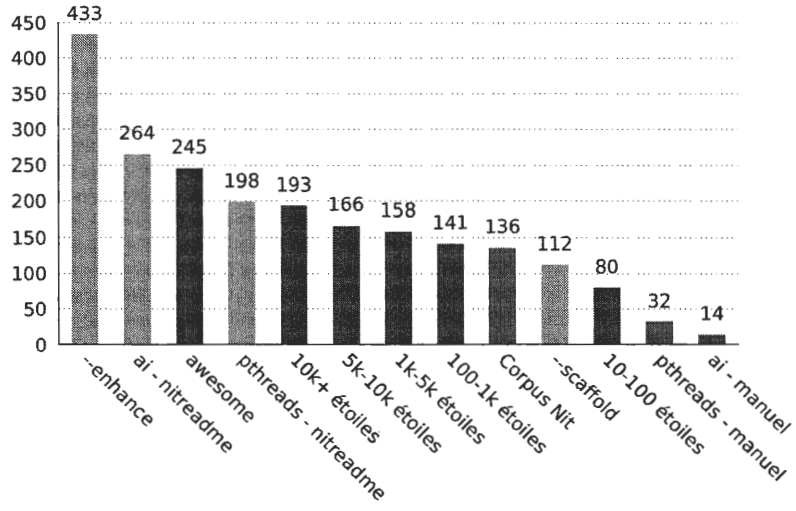


Figure 9.5: Comparaison du nombre de lignes moyen des fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub.

En-têtes de sections La figure 9.6 compare le nombre d'en-têtes de sections dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub.

Le fichier README de **pthreads** (**pthreads - nitreadme** 10 en-têtes) se compare aux projets issus de GitHub comprenant entre 100 et 999 étoiles. Le fichier README de **ai** (**ai - nitreadme** 22 en-têtes) se compare aux projets issus de la liste *awesome*. Dans les deux cas, on constate une nette augmentation par rapport aux versions produites manuellement.

La moyenne pour les fichiers générés est de 5.4 en-têtes par document, ce qui les positionnent en dessous des projets comprenant entre 10 et 99 étoiles (6.7 sections). Les fichiers README améliorés contiennent quant à eux une moyenne de 16.4 sections soit un peu moins que les projets comprenant entre 5 000 et 9 999 étoiles.

Il faut noter ici que nos README ne contiennent pas les trois sections que nous avons systématiquement supprimées : *Problèmes*, *Contribution* et *Licence*, qui sont communes à tous les packages dans le dépôt du projet Nit et donc factorisées dans le README

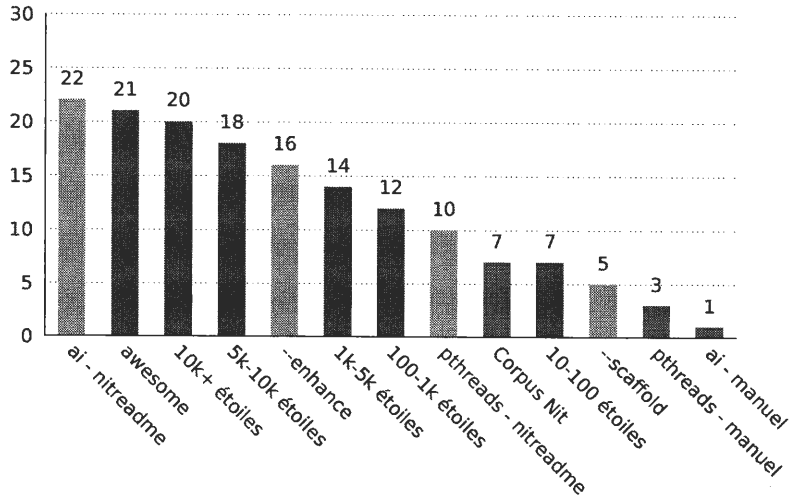


Figure 9.6: Comparaison du nombre d'en-têtes de sections dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub.

principal.

Extraits de code La figure 9.7 compare le nombre d'extraits de code dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub.

Dans tous les cas, on remarque que l'utilisation de l'outil permet d'ajouter des extraits de code là où il n'y en avait aucun auparavant. Cependant, le nombre total d'extraits de code dans le fichier reste inférieur à celui des projets les moins bien notés.

Le nombre moyen d'exemples inclus dans les fichiers générés est seulement de 2.4 par fichier, ce qui est bien en dessous de la moyenne des fichiers du corpus Nit (4.2) et celui de GitHub pour les projets de 10 à 99 étoiles (10.3). Ceci peut s'expliquer par le fait que sur les 50 projets sélectionnées, aucun ne contient de tests unitaires et seulement 16 contiennent des exemples annotés avec `is example`. Les fichiers améliorés contiennent quant à eux beaucoup plus d'exemples avec une moyenne de 10.6 par document, soit

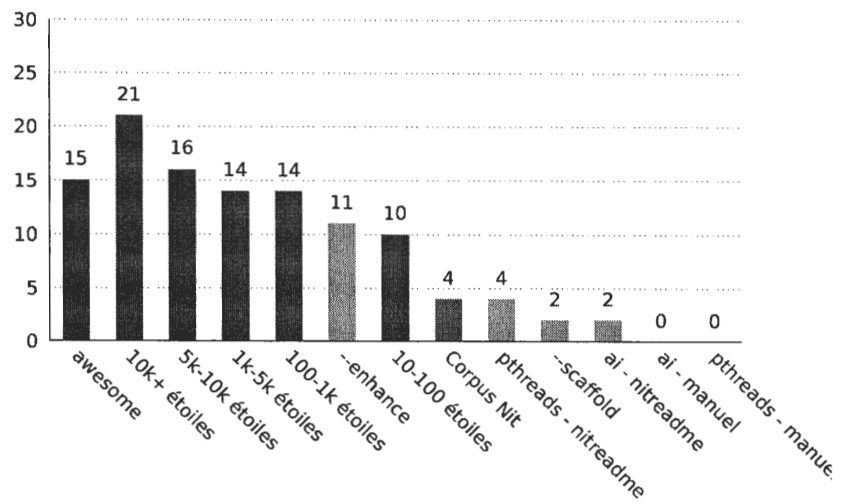


Figure 9.7: Comparaison du nombre de blocs de code dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub.

autant que les projets de 10 à 99 étoiles.

Mises en relief de code Le nombre d'occurrences de la mise en relief de code mesure l'utilisation faite dans le texte des références vers entités de l'API. La figure 9.8 offre une comparaison de ces utilisations dans les fichiers README produits par les experts avec ceux du corpus Nit et ceux issus de GitHub.

Mis à part l'expert 1 dans la documentation de `ai`, les autres fichiers produits avec `nitreadme` comportent plus de références à l'API que les autres fichiers.

Avec une moyenne de 20.9 mises en relief de code par fichier, les fichiers générés par `nitreadme --scaffold` sont comparables à des projets comprenant plus de 10 000 étoiles. La moyenne de 97.1 mises en relief de code par fichier amélioré les place bien au delà de ceux de la liste *awesome* (32.4).

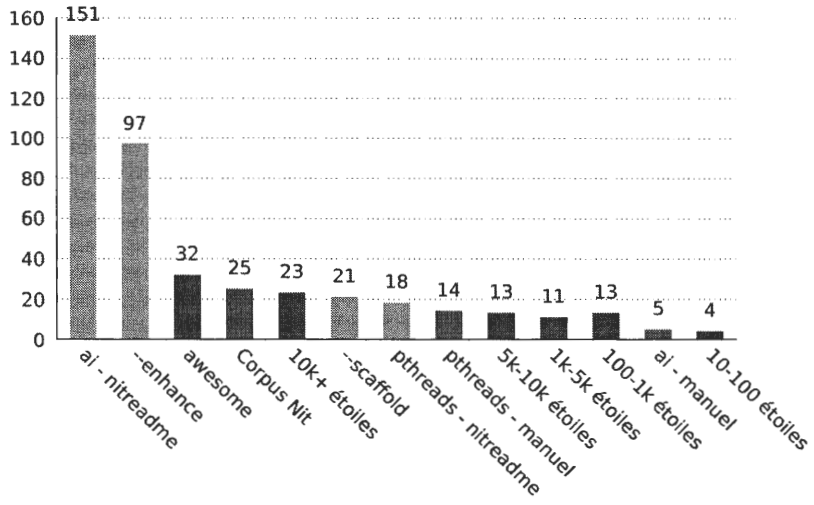


Figure 9.8: Comparaison du nombre de mises en relief de code dans les fichiers README rédigés par les experts avec ceux générés, ceux du corpus Nit et ceux issus de GitHub.

9.5.3 Résultats de la soumission aux experts

La figure 9.9 donne le pourcentage d'acceptation des fichiers générés par l'outil `nitreadme` avec l'option `--scaffold` et améliorés avec `--enhance`.

Pour la génération des nouveaux fichiers, 34 des 50 fichiers (68%) ont été acceptés dans le dépôt sans modification ; 10 fichiers (20%) ont été acceptés après modifications mineures (déplacement, suppression ou ajout de quelques lignes) ; 6 fichiers (12%) ont été rejetés car le résultat n'était pas d'une qualité suffisante.

Pour l'amélioration des fichiers existants, 9 des 18 fichiers (50%) ont été acceptés sans aucune modification ; 5 fichiers (28%) ont été acceptés après des modifications mineures ; 4 fichiers (22%) ont été rejetés.

La figure 9.10 donne le pourcentage d'acceptation pour chaque type de cartes de documentation d'API générées par `nitreadme --scaffold`. Sur les 88 cartes de documentation

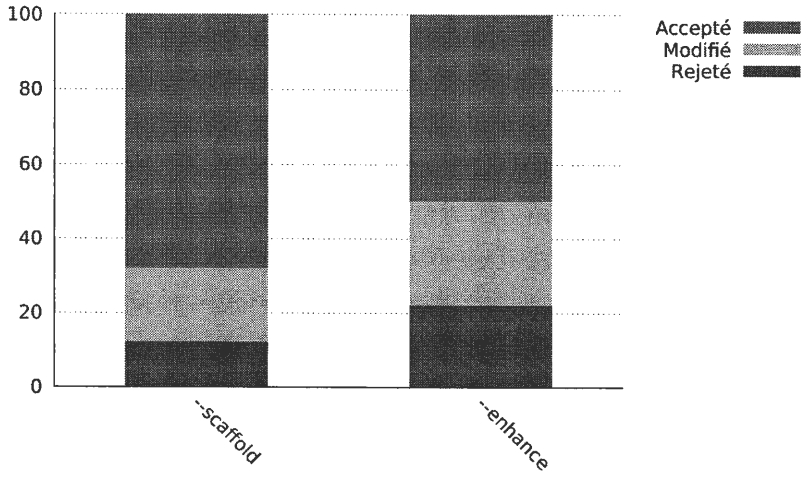


Figure 9.9: Pourcentage de fichiers README générés par `nitreadme --scaffold` et améliorés par `nitreadme --enhance`, qui ont été acceptés, acceptés après modification mineure ou refusés.

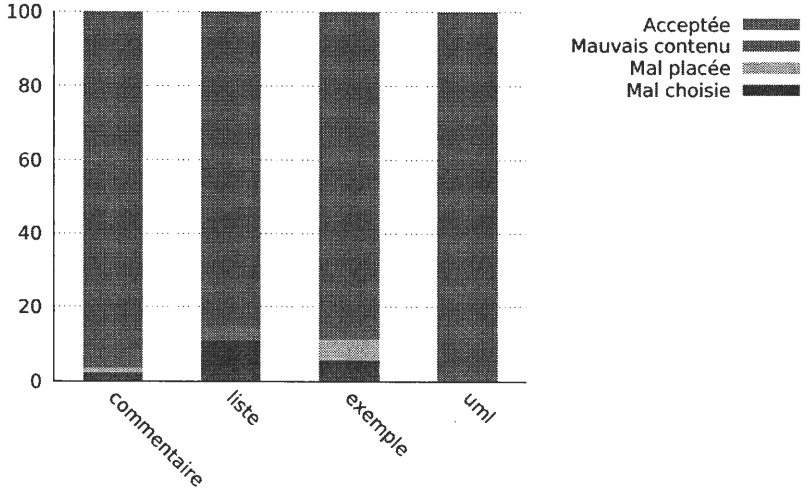


Figure 9.10: Pourcentage de carte de documentation d'API qui ont été acceptées ou refusées (pour diverses raisons) pour `nitreadme --scaffold`.

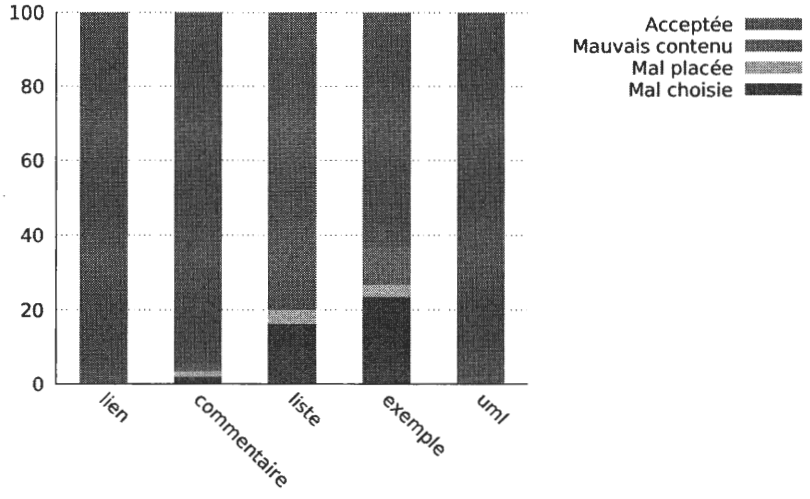


Figure 9.11: Pourcentage de carte de documentation d'API qui ont été acceptées ou refusées (pour diverses raisons) pour `nitreadme --enhance`.

insérées, deux sont jugées mal choisies et une mal placée ; pour sept autres cartes, la carte est bien choisie et bien placée mais le contenu du commentaire à insérer n'est pas correct (commentaire déprécié ou mal formaté). Parmi les 56 listes qui ont été insérées : 6 d'entre elles sont mal choisies (les listes ne contenaient qu'un seul élément) et 2 sont bien choisies et bien placées mais contiennent des entités sans commentaire. Pour les 18 exemples proposés, un seul est considéré mal choisi et un seul mal placé. Les 24 cartes de diagrammes UML ont toutes été acceptées.

La figure 9.11 donne le pourcentage d'acceptation pour chaque type de cartes de documentation d'API ajoutées par `nitreadme --enhance` dans les 18 fichiers existants. Seules deux cartes d'inclusion de commentaires sur les 112 suggérées sont considérées mal choisies ; deux sont mal placées dans le document, deux autres reposent sur un commentaire vide (ne contenant que des espaces ou des sauts de ligne). Quatre listes de propriétés sur les 20 proposées sont rejetées car jugées non pertinentes selon le contexte ; une autre est mal placée. Les suggestions de cartes d'inclusion de code d'exemple sont les plus contestées (11 refusées sur 30 suggérées), souvent car l'exemple est trop long ou complexe pour le contexte où il est proposé. Les 110 cartes de liens et les 19 cartes de

diagrammes UML ont toutes été acceptées sans modification.

9.5.4 Questionnaire de satisfaction

À la fin de la session d'utilisation du mode de suggestion interactif par chaque expert, nous leur avons demandé de répondre à un questionnaire permettant d'évaluer leur expérience avec l'outil. Le questionnaire se présentait sous la forme d'affirmations pour lesquelles l'expert pouvait exprimer son accord ou son désaccord. Ces affirmations sont présentées dans le tableau 9.3. Pour chaque affirmation, la réponse correspondait à une note à attribuer allant de 1 pour *Totalement en désaccord* à 5 pour *Totalement d'accord* ; un champ pour des remarques était aussi disponible sous chaque question. Nous présentons les diverses réponses dans le tableau.

Les notes et remarques dérivées de l'expérimentation avec les experts sont encourageantes. Bien que le système de suggestion ne soit pas optimal, il guide les écrivains dans le choix des sections à ajouter à leurs fichiers README et des entités de l'API à documenter. Il permet de produire une documentation puis maintenir une documentation à moindre effort.

Un constat intéressant se trouve du côté du comportement des experts. En effet, bien que ceux-ci n'aient pas maintenu la bibliothèque à documenter depuis plus d'un an — deux ans dans un cas —, aucun n'a eu recours au code source pour produire son fichier README. Les seuls supports utilisés étaient les suggestions de `nitreadme` et ses liens vers la documentation d'API servie par `nitweb`. Ceci entraîne toutefois un certain risque quant aux entités masquées par l'outil, s'il y en a, puisque celles-ci ne seront alors jamais documentées.

9.5.5 Menaces à la validité de notre étude

Validité de construction La validation est découpée de manière à vérifier chaque mode d'utilisation. Pour chacun, nous validons exactement la dimension qui nous

Tableau 9.3: Questions posées aux experts après leur session d'utilisation du mode de suggestion interactif de l'outil `nitreadme`.

Question	**Moyenne**
L'outil m'aide à trouver quoi documenter. • «Parfois, les suggestions me font perdre le fil de mes pensées.»	3 / 5
Les suggestions faites par l'outil sont pertinentes. • «Il faudrait pouvoir rejeter toutes les suggestions du même type d'un coup.»	4 / 5
L'ordre des sections d'échafaudage est correct.	5 / 5
Les suggestions me permettent d'améliorer le contenu de ma documentation. • «L'outil me fait remarquer les faiblesses de mes commentaires et exemples.» • «J'aimerais pouvoir modifier le contenu des commentaires et exemples importés directement depuis `nitreadme`.»	5 / 5
Elles me permettent de maintenir ma documentation à jour automatiquement.	5 / 5
L'outil est facile à utiliser. • «Il faudrait un système d'auto-complétion pour les commandes et les titres.»	4 / 5
L'outil m'aide à rédiger un meilleur README.	4.5 / 5
Faire la même chose à la main serait plus pénible.	5 / 5

concerne : sa capacité à produire ou à améliorer un fichier README. Nous mesurons cette capacité sur la base d'une augmentation du nombres de lignes, d'en-têtes de sections, d'extraits d'exemples, d'abstractions et de références à la documentation d'API. Nous avons préalablement relié ces caractéristiques à des README de bonne qualité au chapitre 2.

Validité interne La menace majeure à la validité interne de notre étude est le nombre restreint d'experts consultés pour la validation du mode interactif. En effet, ces deux experts ne nous paraissent pas suffisant pour pouvoir juger totalement de l'efficacité de l'outil `nitreadme`. Cependant, ces deux experts sont aussi les auteurs du code et donc les mieux placés pour juger du résultat final produit par l'outil. De plus, ils ont pu comparer leur premier fichier produit manuellement avant de donner leur opinion sur l'outil et d'en noter les dimensions. Nous pensons que, malgré un faible échantillon, nos conclusions démontrent un réel potentiel pour notre approche.

Validité externe Les projets utilisés pour la validation de `nitreadme` dans ses modes interactif (`nitweb/readme`) et d'amélioration (`--enhance`) sont tous issus du corpus utilisé pour développer, tester, puis valider les approches d'alignement de `doc_align` (chap. 8). Ceci engendre un risque de *surajustement* (*overfitting*) de notre approche aux fichiers utilisés pour sa validation.

Le large échantillon de projets sélectionnés pour la validation du mode d'échafaudage (`--scaffold`) nous autorise une certaine confiance quant à la possibilité d'obtenir la même satisfaction pour d'autres projets Nit.

En revanche, les résultats et l'approche pour les produire se basant sur certaines fonctionnalités du langage Nit, nous ne pouvons donc pas généraliser nos observations à des fichiers README pour des projets dans d'autres langages. De plus, nos résultats ne pourraient être généralisés à d'autres types de documentation que les fichiers README.

9.6 Conclusion

Dans ce chapitre, nous avons présenté les rouages internes du processus de suggestion de l'outil `nitreadme`. Nous avons montré que le choix des suggestions s'appuie sur l'alignement du contenu du texte Markdown avec les entités de l'API tel qu'il est créé par l'étape d'analyse précédente, `doc_align`, et la structure traditionnelle du fichier README extraite depuis notre étude empirique du chapitre 2.

À partir de cet alignement, `nitreadme` produit un ensemble de suggestions de cartes de documentation (section 7.3) afin de construire un document contenant les sections traditionnelles du README et des références vers les entités de l'API. Ces suggestions sont sélectionnées en fonction du contexte du curseur. Le curseur représente la position et le contexte de l'écrivain dans son explication. Les modes automatiques pour la génération et l'amélioration des fichiers README sont basés sur les mêmes suggestions que celles du mode interactif mais utilisent un ensemble d'heuristiques afin de choisir les cartes à insérer.

Nous avons validé notre approche et les suggestions proposées aux écrivains dans une suite de trois expérimentations. La validation de notre prototype montre la viabilité de notre approche et l'intérêt d'utiliser `nitreadme` pour la production et la maintenance des fichiers README. Que ce soit dans son mode interactif ou en ligne de commande, il guide l'écrivain dans ses choix de documentation et lui fait prendre conscience des erreurs qu'il peut avoir lui-même commis dans ses commentaires. De plus, la tenue à jour automatique des cartes de documentation insérées permet d'assurer la synchronisation de ce que rédige l'écrivain avec la base de code.

Finalement, signalons que nous avons présenté cette approche à la conférence *AAAI—Statistical Modeling of Natural Software Corpora* à l'hiver 2018 (Terrasa *et al.*, 2018) : « *Using Natural Language Processing for Documentation Assist* ».

CONCLUSION

Le fichier README est le premier — et parfois le seul — artefact de documentation vu par les utilisateurs d'un projet. Comme nous l'avons vu, ce fichier est considéré comme la page de couverture du projet et il est présent dans 99% des projets sur GitHub.

Dans le chapitre 1 de cette thèse, nous avons montré que le README est caractérisé par les mêmes critères de qualité que les autres artefacts de la documentation logicielle. Nous avons déterminé les attributs d'un bon README selon son contenu, sa structure, l'utilisation d'exemples, l'utilisation d'abstractions et sa synchronisation avec le code.

Dans une étude empirique de plus de 500 projets sur GitHub présentée au chapitre 2, nous avons pu regrouper le contenu du README en deux catégories : 1) les sections d'échafaudage qui sont communes à la plupart des fichiers, telles la procédure d'installation, le processus de signalement de bogues ou la licence ; et 2) l'explication de l'API du logiciel qu'il documente, qui contient des listes de classes et de propriétés, la documentation de ces entités, comment les utiliser grâce à des exemples, etc.

Écrire un fichier README à la main est un travail fastidieux dont le résultat doit être maintenu synchronisé avec le code. Or, le contenu du README se base sur ce qui est traditionnellement produit automatiquement par les outils d'auto-documentation d'API tels que Javadoc ou Doxygen. Toutefois, notre revue des générateurs de fichiers README existants au chapitre 3 nous a permis d'identifier tant les limites des solutions existantes que les fonctionnalités nécessaires à l'élaboration d'une solution idéale. Ces fonctionnalités sont la génération assistée de l'échafaudage du fichier README, la génération des sections dédiées à la documentation de l'API, l'annotation, la vérification et l'insertion automatique d'exemples et d'abstractions telles les listes et les figures.

Nous avons montré au chapitre 4 qu'il est difficile de générer automatiquement la structure

du fichier README tout en satisfaisant les exigences des écrivains. C'est pourquoi nous avons proposé une approche *semi-automatisée* d'écriture des fichiers README permettant à l'écrivain d'utiliser la structure qu'il désire mais en y incluant automatiquement les éléments issus de l'auto-documentation d'API.

Afin de mettre en œuvre notre solution, nous avons proposé une spécification de documentation d'API pour le langage Nit nommée `doc_down` présentée au chapitre 5. Cette spécification offre des mécanismes permettant de documenter toutes les entités du code grâce au format Markdown tout en facilitant l'utilisation d'exemples et d'abstractions et en maintenant automatiquement le contenu à jour à l'aide des directives de documentation Nit. Puis, au chapitre 6, nous avons introduit trois outils permettant de générer la documentation d'API en Nit : `nitdoc`, `nitweb` et `nitx`. Ces outils présentaient des points communs que nous avons tâché de factoriser grâce au cadre de développement `doc_commands` pour l'implémentation des directives de documentation Nit. Ces outils et ce cadre de dévevelopement nous ont permis d'égaler — et même dépasser — l'état de l'art des systèmes d'auto-documentation d'API.

Nous avons ensuite présenté la principale contribution de notre travail de thèse dans le chapitre 7 : une approche pour la production de fichiers README axée sur *la suggestion de cartes de documentation* — des extraits de documentation produits par le générateur de documentation d'API, mais pouvant être importés directement dans le corps d'un fichier README — et la conception d'une suite d'outils supportant cette approche, notamment `nitreadme` — pour générer de nouveaux fichiers README à partir du code ou améliorer des fichiers README existants et les tenir synchronisés avec le code — et `nitweb/readme` — pour formuler des suggestions à l'écrivain de façon interactive, pendant la rédaction d'un fichier README.

Nous avons validé l'utilisation du mode de suggestion interactif grâce à la participation de deux experts à qui nous avons demandé de réécrire le fichier README d'une bibliothèque Nit dont ils sont l'auteur, mais cette fois à l'aide du mode interactif de l'outil `nitreadme`. Les fichiers README produits montrent que `nitreadme` permet d'augmenter la structure

et le contenu du README et ce à moindre effort. Et pour valider l'outil en ligne de commande, nous avons généré automatiquement le README de 50 bibliothèques Nit (`--scaffold`) et modifié 18 fichiers README pour des bibliothèques Nit (`--enhance`). Après soumission de ces fichiers aux mainteneurs du projet Nit, 90% des fichiers générés et 78% des fichiers modifiés ont été acceptés pour intégration.

La validation de notre prototype montre la viabilité de notre approche et l'intérêt d'utiliser `nitreadme` pour la production et la maintenance des fichiers README. Que ce soit dans son mode interactif ou en ligne de commande, il guide l'écrivain dans ses choix de documentation et lui fait prendre conscience d'erreurs qu'il peut avoir lui-même commis dans ses commentaires. De plus, la tenue à jour des cartes de documentation utilisées permet d'assurer la synchronisation de ce que rédige l'écrivain avec la base de code.

Principales contributions

Tel qu'évoqué plus haut, la principale contribution de notre travail de thèse se matérialise dans la conception (chap. 7) et la mise en œuvre (chaps. 8 et 9) d'une approche — la *suggestion* de cartes de documentation — et d'une suite d'outils pour la production de fichiers README dont :

- `nitreadme` : un outil de suggestion pour les fichiers README en ligne de commande capable de générer de nouveaux fichiers README à partir du code (approche *a priori*) ou d'améliorer des fichiers README existants et de les tenir synchronisés avec le code (approche *a posteriori*).
- `nitweb/readme` : un outil de rédaction de fichiers README semi-assisté faisant des suggestions à l'écrivain, de façon interactive, pendant la rédaction du fichier README, outil présenté sous la forme d'une extension modulaire au serveur de documentation Nit, `nitweb`.

La mise en œuvre de notre approche repose sur l'*alignement* du contenu du fichier README avec le contenu de l'API qu'il documente (chap. 8), c'est-à-dire l'établissement

de correspondances entre les éléments clés du README et les entités du code source. Ceci permet alors de sélectionner les cartes de documentation pertinentes à insérer ainsi que les positions où les insérer dans le document (chap. 9).

La validation de notre approche représente en elle même une des contributions de notre thèse pour la communauté Nit avec la création d'un nouveau fichier README pour 44 bibliothèques non-documentée jusqu'alors et l'amélioration du README de 14 bibliothèques. Tous ces fichiers ont été acceptés pour intégration au dépôt par les mainteneurs du projet Nit.

Signalons que nous avons présenté cette approche à la conférence *AAAI—Statistical Modeling of Natural Software Corpora* à l'hiver 2018 (Terrasa *et al.*, 2018) : «*Using Natural Language Processing for Documentation Assist*».

Travaux futurs

La présente thèse et le prototype `nitreadme` représentent un premier pas vers une approche efficace pour la création et la maintenance des fichiers README, bien que beaucoup de travail reste à faire. Nos perspectives d'avenir se concentrent tout d'abord sur une validation plus étendue de l'approche en la portant vers un langage plus populaire tel que Java, Ruby ou Python. Ce changement impliquera de revoir certains points clés de notre approche, qui profite pour l'instant des fonctionnalités du langage Nit.

Notre proposition semble compatible avec les techniques de génération automatique de commentaires et d'exemples. Nous nous intéresserons donc à la génération de cartes de documentation dont le contenu est basé sur ces techniques afin de produire des suggestions, et ce même pour du code non documenté. Et maintenant que nous pouvons construire un ensemble de fichiers README utilisant des cartes de documentation, nous nous intéresserons aussi aux capacités de l'apprentissage machine à suggérer des heuristiques intéressantes pour le processus de suggestion. Nos README pourraient alors être utilisés comme jeu d'entraînement (*training set*).

Le README de chaque package Nit représente un élément de documentation qui peut constituer ensuite la documentation d'un projet plus complexe ou d'une bibliothèque standard. Nous souhaitons nous pencher sur la composition d'un manuel multi-pages à l'aide des fichiers README des packages qu'il contient. `nitreadme` permettrait alors de suggérer l'ordre et les relations entre ces pages.

Enfin, et comme l'ont fait remarquer les experts, `nitreadme` pourrait être utilisé comme un outil de modification où l'écrivain pourrait améliorer ses commentaires et ses exemples directement dans l'interface et pousser ensuite les modifications vers son dépôt de sources. Cette fonctionnalité pourrait aussi être ajoutée au serveur de documentation d'API.

ANNEXE A

ANALYSE EMPIRIQUE DU CONTENU DE FICHIERS README PROVENANT DE GITHUB

A.1 Utilisation de blocs Markdown

A.1.1 Utilisation de blocs de citation

L'utilisation des blocs de citation n'est pas fréquente dans notre corpus, avec seulement 116 occurrences soit 0.3 bloc de citation par document. Nous avons repris l'approche d'Ikeda *et al.* pour annoter les blocs de citation. Ainsi, une revue manuelle nous a permis de créer la taxonomie de leurs thèmes d'utilisation. Nous présentons cette taxonomie dans le tableau A.1.

Nous expliquons ces thèmes en fonction de l'utilisation qui en est faite, la liste étant triée selon la position moyenne du thème dans les documents analysés :

- *Statut* : 37 utilisations (31.9%) contenant une note ou un avertissement comme pour le numéro de la version du projet, un projet déprécié, en recherche de mainteneur, ou déplacé ;
- *Fonctionnalité* : 38 utilisations (32.8%) contenant l'objectif du projet ou une fonctionnalité ;
- *Installation* : 4 utilisations (3.4%) contenant un processus d'installation ;
- *Réfs.* : 13 utilisations (11.2%) pour citer un article, un livre ou une personne ;
- *Licence* : 3 utilisations (2.5%) ;

Tableau A.1: Taxonomie des thèmes de blocs de citation les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Statut	37	31.9	0.11	28	8.3	22.0	Version, notes et avertissements
2	Fonct.	38	32.8	0.11	35	10.5	24.8	Explication des fonctionnalités
3	Installation	4	3.4	0.01	3	0.9	32.3	Procédure d'installation
4	Réfs.	13	11.2	0.04	6	1.8	60.6	Citations
5	License	3	2.5	0.01	1	0.3	95.9	Contenu de la licence
-	Autre	21	18.1	0.06	12	3.6	50.1	Accentuation de texte, mauvais usages

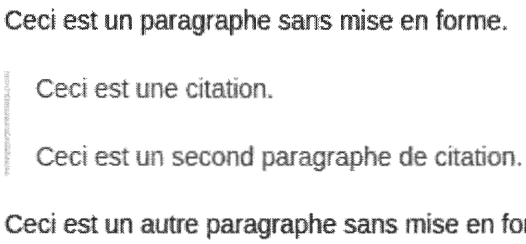


Figure A.1: Bloc de citation transformé en HTML par la plate-forme GitHub.

— *Autre* : 21 autres utilisations (18.1%), par exemple, pour du texte qui n'est pas une citation, un bloc contenant une image ou une erreur dans l'utilisation du format.

On remarque que, dans les faits, les écrivains utilisent souvent le bloc de citation pour accentuer une partie du texte, comme une note importante. Cette utilisation peut s'expliquer par le rendu graphique des blocs de citation par les plate-formes comme GitHub. Comme le montre la capture d'écran en figure A.1 — lequel correspond au bloc de citation présenté ci-haut — le texte écrit comme citation ressort clairement du reste du texte.

Nous avons calculé le nombre moyen de blocs de citation par liste d'origine des projets. Comme le montre la figure A.2, les documents issus des projets de la liste *awesome* contiennent en moyenne plus de blocs de citations que les autres, alors que les projets de la liste *10-100* en contiennent le moins.

A.1.2 Utilisation de blocs de listes

Les listes non-ordonnées sont les plus utilisées avec 1 087 occurrences contre 120 listes ordonnées. Le marqueur de liste le plus fréquent est l'étoile (*) avec 628 occurrences, puis le moins (-) avec 431 occurrences. Le plus (+) est généralement délaissé avec seulement 28 occurrences. Un document comprend en moyenne 3.5 listes.

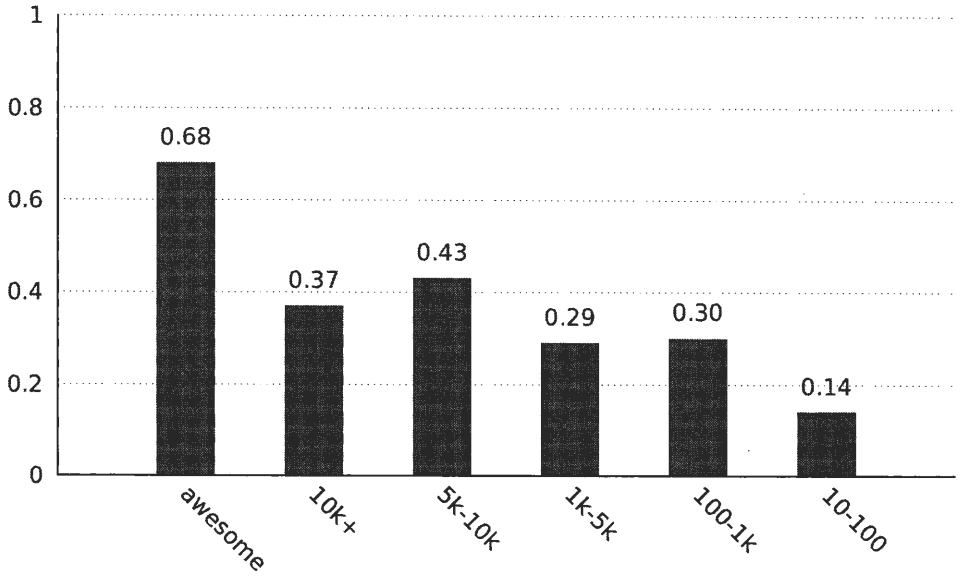


Figure A.2: Fréquence moyenne d'apparition du bloc de citation dans les fichiers README en fonction de la liste d'origine des projets.

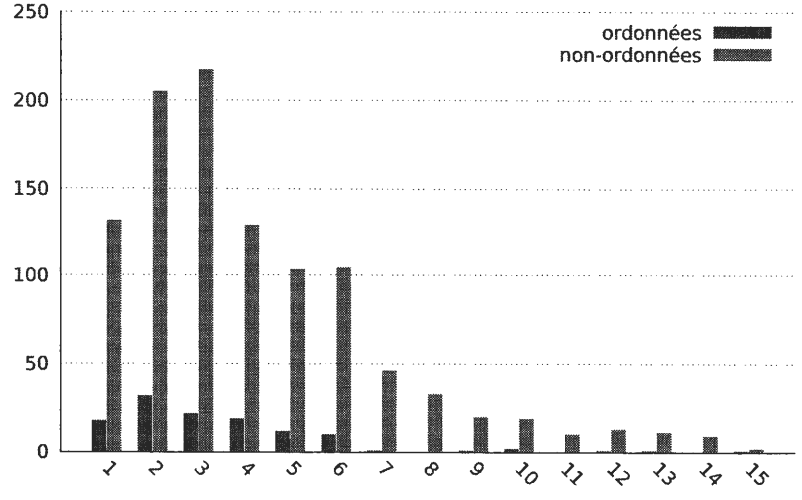


Figure A.3: Nombre d'occurrences de chaque taille de liste trouvée dans le corpus de fichiers README.

Comme le montre la figure A.3, les listes non-ordonnées contiennent généralement entre un (1) et dix (10) éléments avec une moyenne de 4.99 éléments par liste (min. : 1, max. : 123). Les listes ordonnées sont plus courtes, généralement entre un (1) à six (6) éléments, avec une moyenne de 1.75 par liste (min. : 1, max. : 15).

Le tableau A.2 présente la taxonomie des thèmes de listes que nous avons pu observer dans notre corpus. Nous avons catégorisé les listes par une revue manuelle afin de déterminer les utilisations des types de contenu les plus fréquents, puis nous avons utilisé une combinaison d'expressions régulières et de vérifications visuelles pour compter les occurrences.

Nous avons détaillé chacun des thèmes trouvés par ordre d'apparition moyen dans les documents :

- *Sommaire* : 518 tables des matières (33.8%) expliquant le contenu du fichier README ;
- *Installation* : 333 procédures d'installation (21.7%) telles les listes d'étapes d'installation, de pré-requis ou de dépendances ;
- *Utilisation* : 114 procédures de lancement (7.4%) expliquant les étapes de lancement d'un outil ou d'utilisation d'une bibliothèque ;
- *Fonctionnalités* : 168 listes de fonctionnalités (10.9%) expliquant les fonctionnalités couvertes par le projet ou ses objectifs ;
- *Doc.* : 63 listes de liens (4.3%) référençant des ressources externes telles des liens vers des sites internet, fichiers de documentation, blogs ou wikis.
- *API* : 146 listes de référence vers du code (9.5%) faisant directement référence à une entité du code via un identifiant ou une signature ;
- *Auteur* : 82 listes de personnes (5.3%) listant les contributeurs d'un projet soit en référençant un utilisateur GitHub, une adresse e-mail ou un couple nom et prénom ;
- *Version* : 77 listes de changements (5.0%) expliquant les modifications et améliorations entre deux versions du projet ;

Tableau A.2: Taxonomie des thèmes de listes les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Sommaire	518	33.8	1.5	107	32.0	28.5	Table des matières
2	Installation	333	21.7	1.0	156	46.7	32.4	Procédure d'installation
3	Utilisation	114	7.4	0.3	74	22.2	39.5	Procédure d'exécution
4	Fonct.	168	10.9	0.5	95	28.4	51.0	Liste de fonctionnalités
5	Doc	63	4.3	0.2	37	11.1	53.7	Références externes
6	API	146	9.5	0.4	69	20.7	56.2	Références vers du code
7	Auteur	82	5.3	0.2	47	14.1	69.9	Noms propres
8	Version	77	5.0	0.2	56	16.8	80.7	Liste de changements
-	Autre	31	2.0	0.1	29	8.7	52.4	Autres types de listes

— *Autre* : 31 autres listes (2.0%) comprenant aussi les mauvaises utilisations du format.

Bien que les sommaires ne figurent pas parmi les thèmes d'Ikeda *et al.* (2018), nous avons dénombré au total 107 documents comportant des tables des matières. La quantité de sommaires peut sembler énorme au premier abord, mais la structure du format Markdown fait en sorte que les sous-listes sont comptées comme des listes à part entière. De plus, en fonction des blocs utilisés dans les listes ou des sauts de lignes, une simple liste peut être composée de plusieurs listes Markdown. Le nombre de 518 occurrences correspond donc à la somme des listes de premier niveau et de leurs sous-listes.

La structure des sections d'un document peut en effet être représentée comme des listes imbriquées. Le fait que les sections Markdown ne soient pas numérotées peut expliquer la préférence des écrivains pour les listes non-ordonnées. On notera tout de même que près d'un tiers des listes ordonnées utilisées le sont pour des tables des matières. Dans de rares cas (11 documents), les écrivains ont pris la peine de numéroter les sections à la main, auquel cas la numérotation correspond à celle de la table des matières.

Les procédures d'installation et de lancement sont généralement présentées par étapes, auquel cas les listes sont utiles. Bien que les listes non-ordonnées soient là aussi préférées, les étapes d'installation et de lancement représentent les autres utilisations des listes ordonnées correspondant à la numérotation des étapes.

Les listes relatives à l'installation sont composées des listes de dépendances (41 listes de dépendances sur les 267). Ici, les écrivains préfèrent aussi les listes non-ordonnées.

Les fonctionnalités des projets sont présentées sous la forme de listes non-ordonnées (120 occurrences). Les listes sont aussi utilisées pour faire références aux éléments du code comme les packages, classes ou méthodes (83 occurrences).

Enfin, les listes sont utilisées pour la présentation des auteurs et mainteneurs des projets, pour la présentation des liens ou citations de ressources externes (par ex., documentation, documentation d'API, livres ou articles), ou encore pour lister les changements opérés

entre deux versions (*changelog*).

A.1.3 Utilisation de blocs de code

Le corpus comprend un total de 2 122 blocs de code ce qui représente une moyenne de 6.3 blocs de code par document. On remarque tout d'abord que la notation avec barrière est généralement préférée par les écrivains avec 1 745 occurrences (82.2%) contre 377 (17.8%) pour la notation indentée. Cette préférence peut être expliquée par le fait que GitHub colore syntaxiquement le code source si un langage est indiqué.

La répartition des langages utilisés dans les blocs de code avec barrières est donnée en figure A.4. Shell est le langage le plus fréquent avec 391 occurrences (22.4%) ; viennent ensuite JavaScript (345 occurrences, 19.7%), puis Java (101 occurrences, 5.7%). Étrangement, dans 270 cas, les écrivains utilisent un bloc de code avec barrières mais ne précisent pas le langage utilisé.

Tous types et langages confondus, les blocs de code sont utilisés pour présenter des instructions d'installation, de lancement ainsi que des exemples d'utilisation de code, comme l'indique le tableau A.3.

Les procédures d'installation (*Installation*) totalisent 762 occurrences (35.9% du total). Elles ont pour objectif d'expliquer comment récupérer les sources du projet et les compiler ou comment installer l'outil. Pour les isoler, nous avons repéré les commandes de compilation et d'installation telles `make`, `sudo`, `git clone` ou `npm install`. Cette même technique est utilisée par Hassan et Wang (2017) mais limitée aux outils relatifs à Java.

Les procédures d'exécution (*Exécution*) comptent 403 occurrences (18.9%). Elles permettent d'expliquer comment lancer un outil. Pour les isoler, nous avons repéré les commandes permettant d'exécuter un programme telles que celles préfixées par « `./` », ainsi que celles spécifiant un langage et son interpréteur (e.g., `python script.py`) ou un outil (e.g., `npm start`).

Tableau A.3: Taxonomie des thèmes des blocs de code les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Installation	762	35.9	2.3	211	63.2	36.6	Procédures d'installation
2	Exécution	403	18.9	1.2	131	39.2	43.0	Procédures d'exécution
3	Utilisation	886	41.7	2.6	146	43.7	54.1	Exemples de code
-	Autre	71	3.3	0.2	32	9.6	56.5	Autre chose que du code

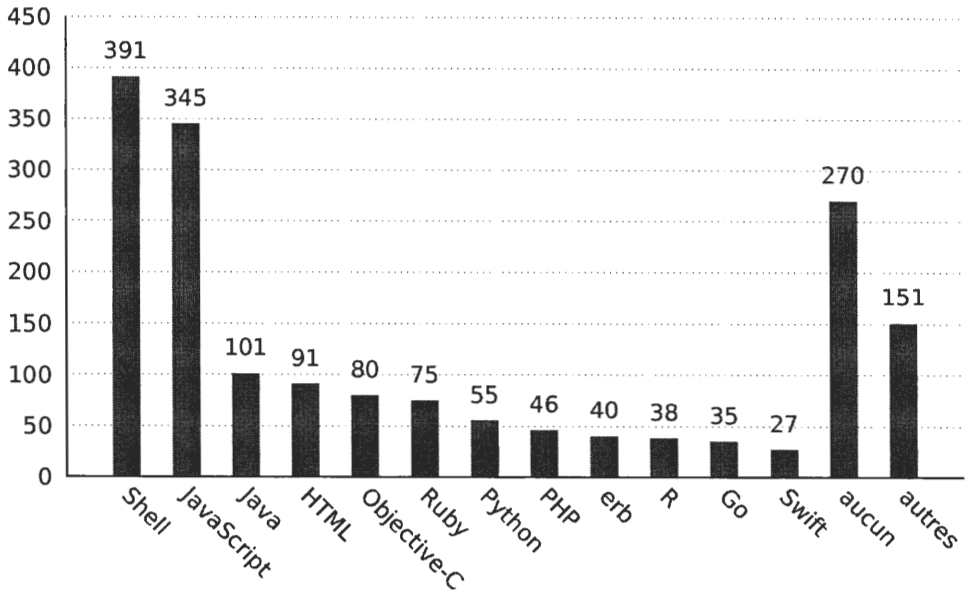


Figure A.4: Nombre d'occurrences des langages indiqués dans les blocs de code avec barrières du corpus de fichiers README.

Fait intéressant, la quasi-totalité des extraits de code identifiés comme du Shell correspondent à des procédures d'installation et d'utilisation (387/391 blocs Shell). Les procédures sont donc généralement données pour des systèmes Unix si l'on compare l'utilisation du langage Shell aux deux (2) seules occurrences du langage Bat.

L'autre part importante des utilisations de blocs de code est la présentation d'exemples d'utilisation (*Utilisation*), ou comment importer et utiliser une bibliothèque ou étendre les fonctionnalités d'un outil. Nous les avons séparé volontairement des thèmes *Installation* et *Exécution* afin de montrer la quantité d'exemples purs dans notre corpus. Ces exemples sont comptabilisés manuellement et représentent 886 occurrences (41.7%).

Enfin, *Autre* regroupe les 71 (3.3%) blocs de code ne contenant pas de code. Dans ces cas, le bloc de code peut être utilisé pour la mise en forme à chasse fixe qui lui est associée, par exemple pour présenter le contenu d'un fichier de licence, dessiner une structure de données ou écrire du texte.

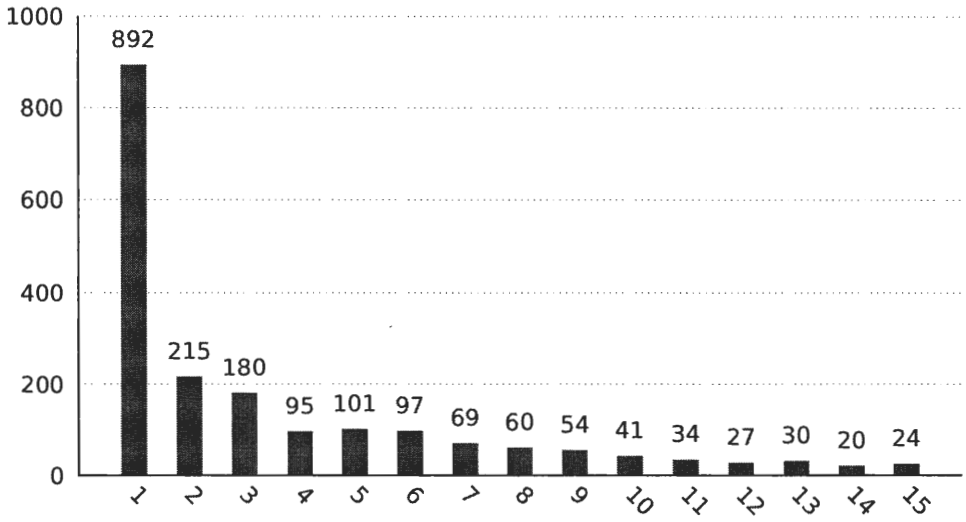


Figure A.5: Fréquence du nombre de lignes par bloc de code dans la totalité du corpus de fichiers README.

La figure A.5 présente la fréquence du nombre de lignes par bloc de code sur la totalité du corpus analysé. Les blocs sont généralement courts : 70% d'entre eux sont composés de 5 lignes ou moins et 95% font 20 lignes ou moins. La moyenne est de 5.7 lignes de code par bloc (min. : 1, max. : 149). Nous n'observons pas de corrélation entre le nombre de lignes de code par bloc et le thème du bloc, ni avec la quantité d'étoiles du projet.

Si on compare les positions moyennes des différents thèmes, on remarque que les exemples d'installation arrivent en premier (position moyenne à 36% du document). Suivent les exemples d'exécution (43% du document) et, enfin, les exemples d'utilisation (54% du document).

Comme le montre l'histogramme présenté en figure A.6, les projets de la liste *awesome* possèdent le plus de blocs de code relatifs à l'installation. Si l'on compare le nombre d'étoiles, les projets avec le plus d'étoiles sont mieux documentés quant à leurs procédures d'installation.

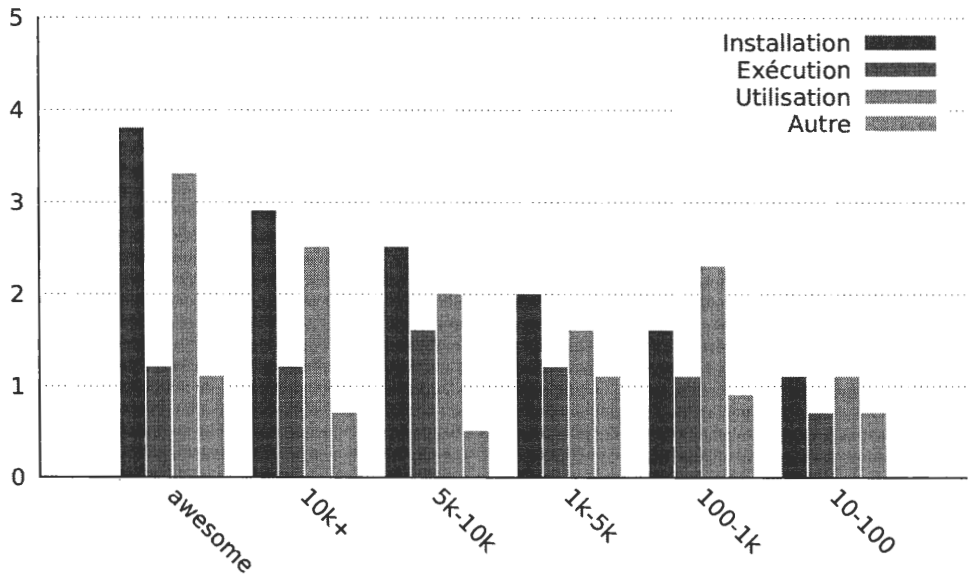


Figure A.6: Fréquence d'apparition de chaque thème de bloc de code en fonction des listes de projets de notre corpus.

Ikeda *et al.* (2018) observent que les titres en rapport avec l'exécution des projets sont moins nombreux pour les bibliothèques que pour les applications. Les résultats que nous présentons pour les blocs de code sont cohérents avec cette observation. En effet, l'histogramme en figure A.7 montre la répartition des thèmes de blocs de code en fonction du type de projet analysé. On remarque que les blocs de code relatifs à l'exécution sont moins fréquents pour les bibliothèques. En revanche, la quantité d'exemples de code augmente pour ce type de projet. Les langages, type de projet à part, proposent plus d'instructions relatives à l'installation que tout autre type de bloc de code.

Tableau A.4: Taxonomie des thèmes de liens les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Badge	675	8.7	2.0	201	60.2	20.5	Badges
2	Sommaire	755	9.7	2.3	66	19.8	21.0	Liens internes
3	Site off.	817	10.5	2.4	199	59.6	27.8	Pages du site officiel
4	Installation	297	3.8	0.9	132	39.5	34.4	Binaires ou archives
5	Dépôts	630	8.1	1.9	169	50.6	36.1	Liens vers des dépôts GitHub
6	Doc.	1287	16.6	3.9	287	85.9	47.5	Documentation
7	Démo.	345	4.4	1.0	142	42.5	48.2	Liens vers une démonstration
8	API	1286	16.5	3.9	186	55.7	49.6	Liens vers le code
9	Problèmes	299	3.8	0.9	147	44.0	57.6	Issues ou aide
10	Licence	147	1.9	0.4	122	36.5	69.9	Liens vers la licence
11	Auteur	522	6.7	1.6	168	50.3	71.5	Noms, adresses courriels
-	Autre	716	9.2	2.1	172	51.5	51.6	Dont les liens brisés

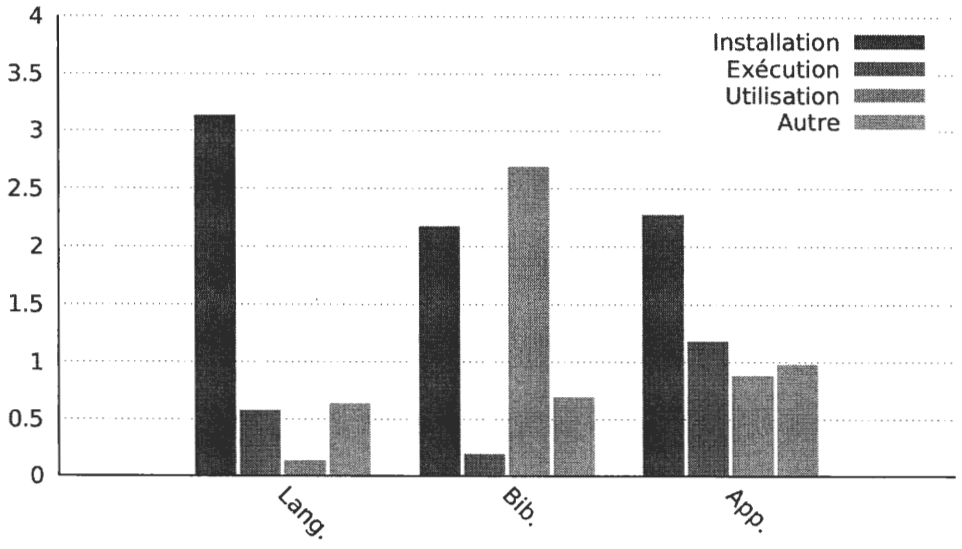


Figure A.7: Fréquence d'apparition de chaque thème de bloc de code par rapport au type de projet.

A.2 Utilisation de la syntaxe en ligne

A.2.1 Utilisation de liens

Le corpus contient 7 776 utilisations de liens dans le Markdown, soit une moyenne de 23.1 par document. Nous avons établi la liste des thèmes de liens par une première revue visuelle afin de déterminer les utilisations les plus fréquentes. Nous avons ensuite utilisé des expressions régulières sur les adresses suivi d'une vérification visuelle pour compter les occurrences. Le tableau A.4 présente la taxonomie des thèmes de liens trouvés dans le corpus, dont nous décrivons certaines caractéristiques ci-bas, par ordre d'apparition du thème dans le README.

Badge Généralement utilisé au tout début du document. Les badges sont identifiés grâce à la présence des mots `ci` (pour `continuous integration`), `badge` et `shield`

dans l'adresse du lien. Comme nous le verrons dans la prochaine sous-section, les badges sont majoritairement utilisés en conjonction avec une image.

Sommaire Dénote des liens internes au document README (ou ancres), lesquels permettent de sauter directement à une section du document et permettent de composer les tables des matières (Github, Inc., 2018a). Ils sont détectés, entre autres, par la présence du préfixe «#» dans l'URL et apparaissent au début du document juste après les badges.

Site off. Regroupe les références vers le site officiel de l'outil ou de la bibliothèque, et sont identifiés grâce à la valeur de la clé `repo.homepage` fournie par les auteurs du projet sur GitHub.

Installation Permettent de télécharger l'outil ou la bibliothèque présentée par le dépôt.

Dépôts Pointent vers des adresses de dépôts sur GitHub, souvent utilisées pour présenter les dépendances d'un projet.

Doc. Thème le plus fréquent, pour des liens vers les ressources de documentation tels les blogs, les wikis et les documentations d'API externes au README. Ces liens ont été isolés par des expressions régulières sur des mots tels `doc`, `api`, `blog` ou `wiki` puis une revue visuelle.

API Pointent vers du code source. Ils correspondent à des fichiers dans le dépôt ou d'autres plate-formes spécialisées comme *jsFiddle*[1], un outil d'échange de code JavaScript (*Code*).

Autre Regroupe les liens (9.2%) que nous n'avons pas réussi à catégoriser, dont 283 liens (3.6%) proposant une adresse absolue mais inaccessibles lorsque testées avec `curl`[2].

---

1. `https://jsfiddle.net/`

2. `curl` est un outil de transfert de fichiers en ligne de commande supportant les requêtes HTTP — `https://curl.haxx.se/`.

Tableau A.5: Taxonomie des thèmes d'images les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Logo	65	4.9	0.2	63	18.9	5.0	Logos de projets
2	Badge	1025	76.8	3.1	226	67.7	15.4	Badges
3	Démo.	197	14.8	0.6	44	13.2	44.0	Exemples de sorties, captures d'écrans, démonstrations
-	Autre	48	3.6	0.1	89	26.6	43.6	Avatars, illustrations, mêmes...

A.2.2 Utilisation d'images

Nous avons catégorisé les images par une revue manuelle afin de déterminer les utilisations les plus fréquentes, puis nous avons utilisé des expressions régulières sur les adresses et textes alternatifs suivi d'une vérification visuelle pour compter les occurrences. Le tableau A.5 présente la taxonomie des thèmes d'images établie à partir de notre corpus.

Logo Les logos de projets sont plus rares que nous ne l'attendions avec seulement 4.9% d'utilisation. Lorsque présents, ils le sont généralement au tout début du document (**% pos.** = 5.0).

Badge Ces images sont celles les plus souvent utilisées. Elles sont identifiées grâce

à la présence des mots `ci` (`continuous integration`), `badge` et `shield` dans l'adresse de l'image ou son texte alternatif.

Démo. Ces images, les deuxiémes plus utilisées et qui ont été comptées manuellement, correspondent à des captures d'écrans des programmes et d'exemples de clients des bibliothèques. 14.8% des images montrent des exemples de fonctionnalités ou résultats, parfois même à l'aide de fichiers *gif* animés.

Autre Parmi les autres types d'images, nous avons pu observer des avatars de développeurs, des illustrations ou encore des *memes*[3] ; 30 images (2.2%) comportaient des liens erronés dont nous n'avons pas pu trouver le contenu.

Seulement quatre (0.3%) images correspondent à des diagrammes et toutes sont utilisées pour représenter des processus. Pourtant, la présence d'abstraction telles des diagrammes UML fait partie des caractéristiques d'une bonne documentation. On peut donc se demander pourquoi les écrivains n'en font pas usage.

En outre, seulement 215 images sur 1 335 (16.1%) sont hébergées sur GitHub, le reste provenant de ressources externes telles des sites officiels des projets, des wikis ou des sites de partage d'images ou de vidéos. Il ne semble donc pas coutume de partager les ressources avec le fichier README, peut-être pour des raisons pratiques comme le temps de transfert ou encore les droits d'auteurs, par exemple.

A.2.3 Utilisation de mises en relief de texte

Avec 1 628 utilisations (moy. : 4.9 par document), les mises en relief fortes sont trois fois plus présentes dans notre corpus que les légères, qui ne comptent que 537 utilisations (moy. : 1.6 par document).

Nous avons catégorisé les mises en relief par une revue manuelle afin de déterminer les utilisations les plus fréquentes, puis nous avons utilisé des expressions régulières et une

3. `https://en.wikipedia.org/wiki/Meme`

vérification visuelle pour compter les occurrences.

Le tableau A.6 présente la taxonomie des thèmes d'utilisation de la mise en relief forte dans notre corpus. Celles-ci sont majoritairement utilisées pour présenter des noms d'auteurs ou de contributeurs comme `**Alexandre Terrasa**`; parfois, seule une adresse e-mail est présente; parfois encore, le nom est accompagné de l'adresse e-mail. Nous avons regroupé les 805 utilisations (59.4%) de ce type sous le thème *Auteur*.

Le thème *Code* contient les identifiants issus du code tels les noms de classes ou de méthodes, les signatures ou encore les noms d'options, qui représentent 381 occurrences (23.4%). Nous avons isolé ces entités en comparant les occurrences avec le contenu des fichiers source du dépôt et avec une passe visuelle.

La troisième catégorie d'utilisation, *Texte*, correspond à des mots de la langue anglaise dans 291 cas (17.9%). Les mots sont corrélés avec le contenu du dictionnaire Debian[4].

Les 103 utilisations du thème *Autre* (6.3%) regroupent des numéros de version, des chemins, des dates ou encore des chiffres.

Le tableau A.7 présente la taxonomie des thèmes des mises en relief légères. Par ordre d'apparition dans les documents, on remarque que les numéros de version sont plus vers le début du document, suivis des noms d'auteurs.

C'est la présentation de texte en langue naturelle qui est la plus fréquente, avec 403 occurrences (*Texte*, 75.0%). Parmi celles-ci, 271 concernent plus d'un mot et 132 un seul mot. Le code et les identifiants représentent l'autre type d'utilisation avec 117 occurrences (*Code*, 21.8%). On retrouve ces constructions tout au long du document.

Tableau A.6: Taxonomie des thèmes de mises en relief fortes les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Version	48	2.9	0.14	25	7.5	25.5	Numéros de version
2	Auteur	805	49.4	2.41	141	42.2	32.0	Noms propres
3	Texte	291	17.9	0.87	88	26.3	43.1	Mots de la langue naturelle
4	Code	381	23.4	1.14	91	27.2	54.6	Identifiants, expressions, chemins
-	Autre	103	6.3	0.31	49	14.7	36.9	Dates, chiffres,...

Tableau A.7: Taxonomie des thèmes de mises en relief légères les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Version	11	2.0	0.1	8	2.4	29.3	Numéros de version
2	Auteur	6	1.1	0.1	6	1.8	30.7	Noms propres
3	Texte	403	75.0	1.2	153	45.8	49.2	Mots de la langue naturelle
4	Code	117	21.8	0.4	43	12.9	55.3	Identifiants, expressions, chemins

Tableau A.8: Taxonomie des thèmes de mises en relief de code les plus fréquents dans les fichiers README.

Légende: **occ.**: nombre d'occurrences du thème dans le corpus; **% occ.**: pourcentage d'occurrences associées à ce thème; **moy.**: moyenne du nombre d'occurrences du thème par document; **# doc.**: nombre de documents contenant le thème; **% doc.**: pourcentage de documents contenant le thème; **% pos.**: position moyenne du thème dans les documents en pourcentage de la taille du document. Les thèmes sont triés par position moyenne dans les documents.

#	Thème	occ.	% occ.	moy.	# doc.	% doc.	% pos.	desc.
1	Version	34	0.5	0.1	17	5.1	43.6	Numéros de version
2	Code	6909	98.5	20.7	236	70.7	51.4	Noms propres
-	Autre	69	1.0	0.2	25	7.5	50.8	Identifiants, expressions, chemins

A.2.4 Utilisation de mises en relief de code

Nous avons dénombré 7 012 utilisations de la mise en relief de code dans notre corpus, soit une moyenne de 20.8 par document. Le tableau A.8 donne le nombre d'occurrences de chaque thème détecté.

Comme on pouvait s'y attendre, les mises en relief de code sont généralement utilisées pour présenter du code (6909 occurrences). Parmi ces utilisations, on peut distinguer :

— 3949 références au code, correspondant à des identifiants comme `‘MaClasse‘` (56.3% des utilisations).

— 1 255 occurrences (17.9%) d'extraits de code, utilisés pour montrer des instructions ou des expressions plus complexes qu'un simple identifiant. Certaines expressions

4. Fichier `/etc/share/dict`

sont syntaxiquement correctes — par ex., `'x < 2'` — alors que d'autres sont à comprendre en fonction du texte trouvé autour de la mise en relief comme `'< 2'`. Ces expressions comprennent aussi la présentation d'extraits de code HTML.

— 955 occurrences (13.6%) de noms de commandes à taper dans le terminal par exemple `'bower install'` ou des noms d'options comme `'-help'`.

— 750 occurrences (10.7%) pour des noms de fichiers, chemins et URLs.

Les numéros de version sont parfois présentés dans des mises en relief de code (34 occurrences, 0.48%). Enfin, le thème *Autre* regroupe toutes les utilisations de mots sans rapport avec le code comme des mots ou des références à des sections du document (69 occurrences, 1.0%).

Les mises en relief de code sont donc généralement utilisées pour montrer un lien entre la langue naturelle écrite dans le fichier README et des entités du code source que ce fichier documente.

Avec une position moyenne autour de 50%, on peut remarquer que les mises en relief de code sont utilisées tout au long du document, en général vers le milieu du document sans rapport avec le thème de la mise en relief.

ANNEXE B

DIRECTIVES DE DOCUMENTATIONS NIT

Tableau B.1: Directives de documentation d'API.

`sign`	Signature de l'entité comprenant les noms et les types
`synopsis`	Première lignes du commentaire de l'entité
`comment`	Lignes du commentaire de l'entité sauf la première
`doc`	Lignes du commentaire de l'entité
`code`	Code source rattaché à l'entité

Tableau B.2: Directives pour la structuration de la documentation.

`include`	Inclue un fichier externe
`toc`	Table des matières du fichier README du commentaire de l'entité ou du fichier Markdown passé en paramètre
`link`	Lien vers l'entité passée en paramètre dans l'auto-documentation d'API. Permet aussi de créer un lien vers des sections de la table des matières.

Tableau B.3: Directives de listes et d'abstractions.

`examples`	Liste des exemples associés à l'entité
`tests`	Liste des tests unitaires associés à l'entité
`parents`	Liste des ancêtres directs de l'entité
`ancestors`	Liste des ancêtres directs et indirects de l'entité
`children`	Liste des descendants directs de l'entité
`descendants`	Liste des descendants directs et indirects de l'entité
`features`	Liste des fonctionnalités *intéressantes* de l'entité
`defs`	Liste de toutes les définitions contenues dans l'entité
`intros`	Liste des entités introduites dans l'entité
`redefs`	Liste des entités redéfinies dans l'entité
`lin`	Liste linéarisée des définitions de l'entité
`new`	Liste des entités appelant le constructeur de la classe passée en paramètre
`call`	Liste des entités appelant à la méthode passée en paramètre
`param`	Liste des méthodes acceptant comme paramètre le type passé en paramètre
`return`	Liste des méthode retournant le type passé en paramètre
`contributing`	Liste des contributions de l'utilisateur passé en paramètre
`maintaining`	Liste des projets maintenus par l'utilisateur passé en paramètre
`tag`	Liste des projets associés au mot-clé passé en paramètre
`all`	Liste toutes les entités contenues dans le modèle Nit
`list`	Liste manuelle d'entités à partir de l'index Nit
`random`	Liste d'entités choisies aléatoirement
`inh`	Graphe d'importation ou d'héritage à partir de l'entité
`uml`	Diagramme UML à partir de l'entité

Tableau B.4: Directives liées à la documentation des packages.

`metadata`	Liste des méta-données de l'entité depuis le fichier `package.ini`
`ini-name`	Nom de l'entité depuis le fichier `package.ini`
`ini-desc`	Description courte de l'entité depuis le fichier `package.ini`
`ini-git`	URL du dépôt Git depuis le fichier `package.ini`
`ini-license`	Licence depuis le fichier `package.ini`
`ini-issues`	URL du gestionnaires de problèmes depuis le fichier `package.ini`
`ini-maintainer`	Mainteneur principal du projet depuis le fichier `package.ini`
`ini-contributors`	Liste des contributeurs depuis le fichier `package.ini`
`ini-tags`	Liste des mot-clés depuis le fichier `package.ini`
`git-clone`	Commande d'obtention des sources de l'entité avec Git
`testing`	Commande de vérification des tests unitaires avec `nitunit`
`mains`	Liste des exécutables contenus dans l'entité
`main-compile`	Commande de compilation pour l'entité avec `nitc` (si exécutable)
`main-run`	Commande de lancement pour l'entité (si exécutable)
`man-file`	Page MAN de l'entité (si exécutable)
`man-synopsis`	Synopsis issu de la page MAN de l'entité (si exécutable)
`man-opts`	Liste des options issues de la page MAN de l'entité (si exécutable)
`contrib-file`	Chemin vers le fichier `CONTRIBUTING.md` associé à l'entité
`contrib-file-content`	Contenu du fichier `CONTRIBUTING.md` associé à l'entité
`license-file`	Chemin vers le fichier `LICENSE.md` associé à l'entité
`license-file-content`	Contenu du fichier `LICENSE.md` associé à l'entité

ANNEXE C

CAPTURES D'ÉCRAN DES OUTILS DE DOCUMENTATION D'API NIT

Figure C.1: Résultat de la commande `children: markdown::MdRenderer` tel que présenté par `nitx`.

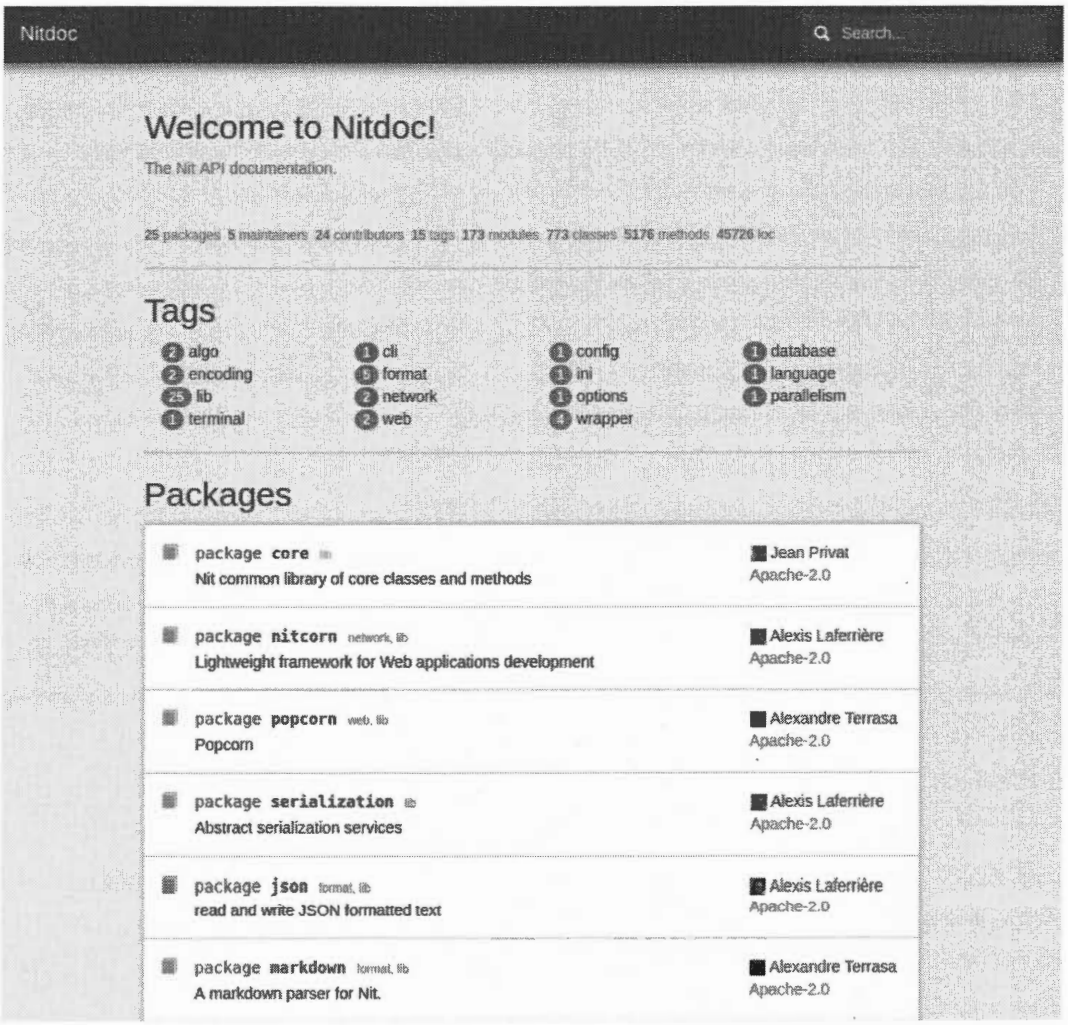


Figure C.2: Capture d'écran de la page d'accueil de la documentation générée par `nitdoc` avec son catalogue de packages.

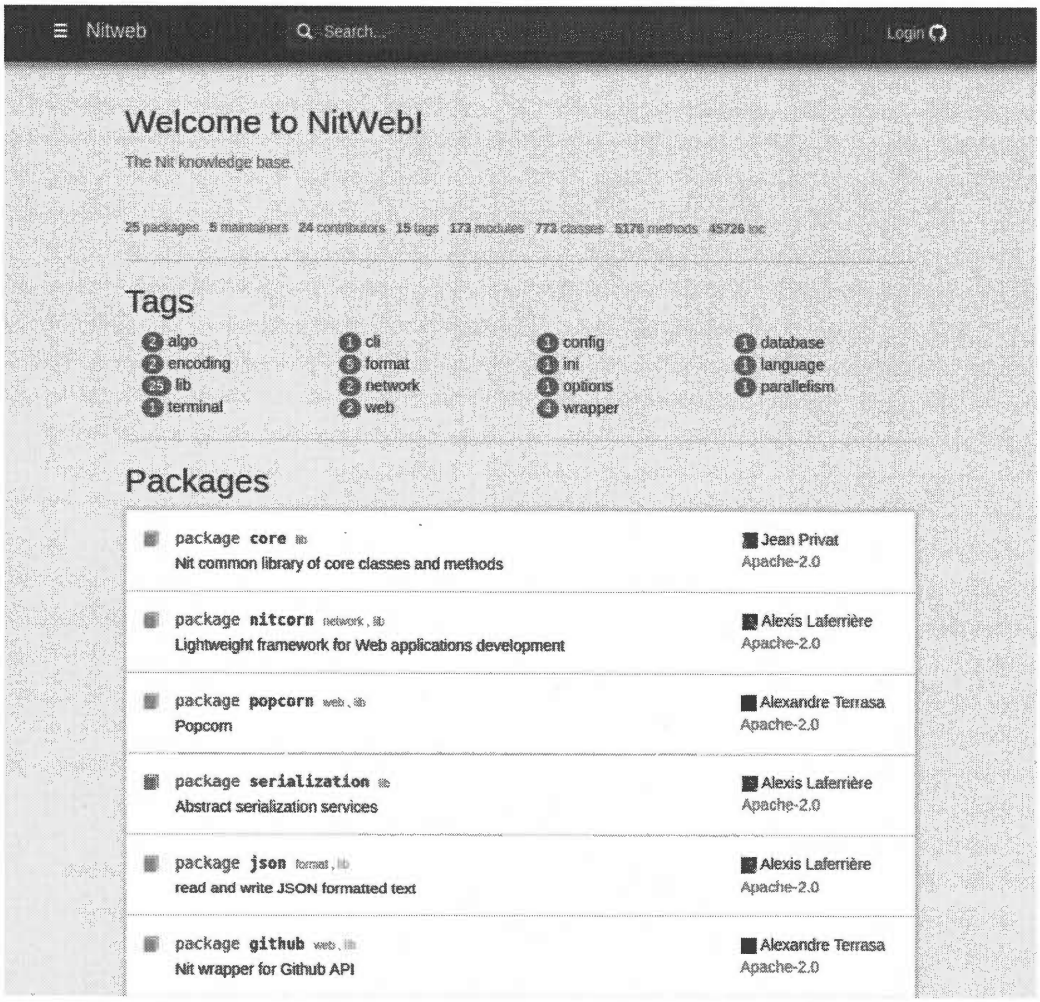


Figure C.3: Capture d'écran de l'interface d'utilisation HTML du serveur de documentation `nitweb`.

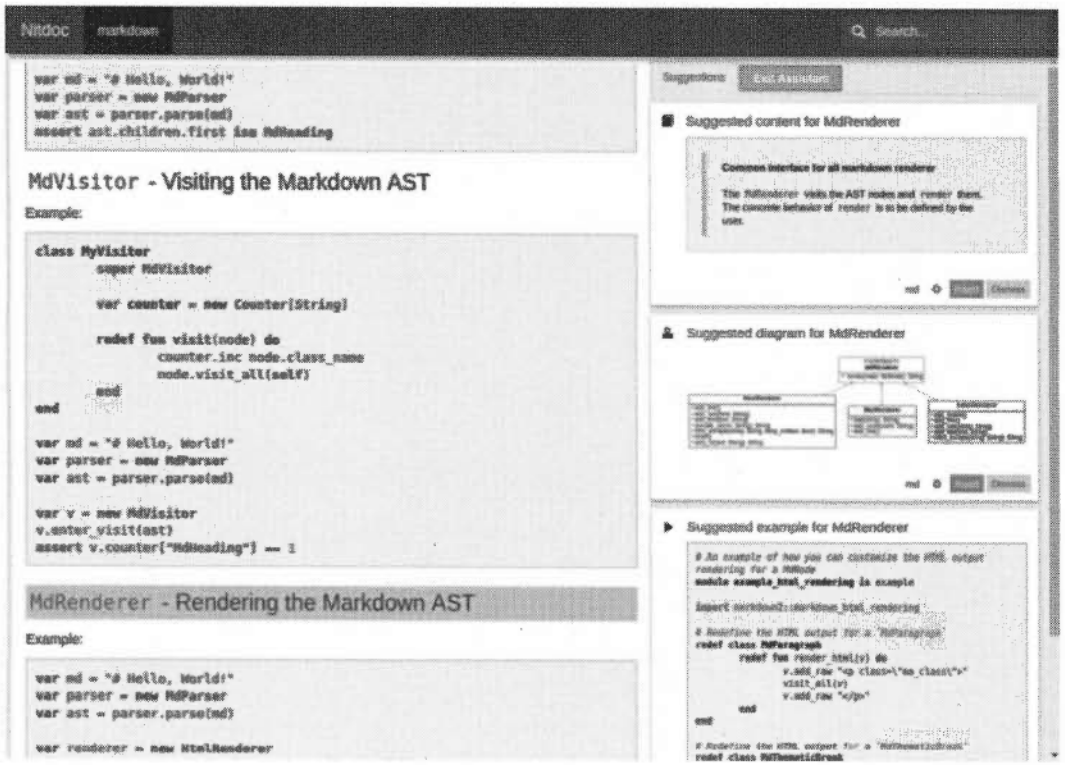


Figure C.4: Capture d'écran de l'interface d'utilisation de l'outil `nitweb/readit`.

ANNEXE D

INTERFACE DE L'OUTIL GOOGLE SPREADSHEET EXPLORE

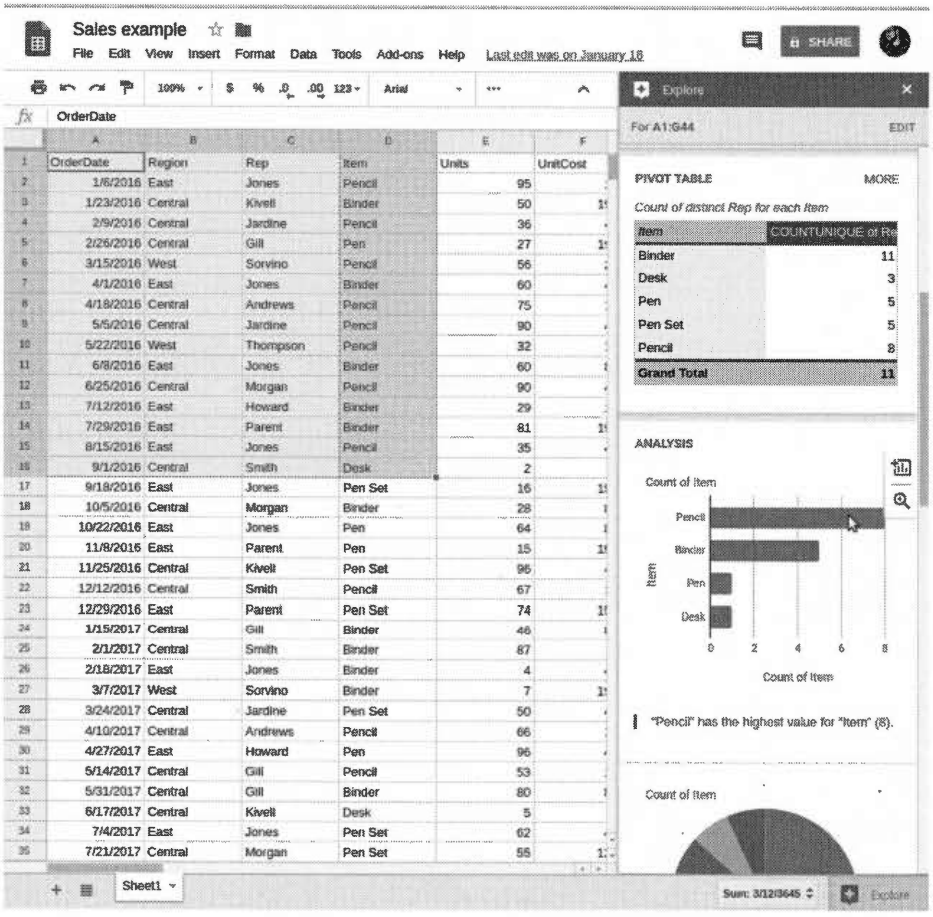


Figure D.1: Cartes de suggestions présentées par l'outil `Google SpreadSheet Explore`.

ANNEXE E

CARTES DE DOCUMENTATIONS SUGGÉRÉES PAR `NITREADME`

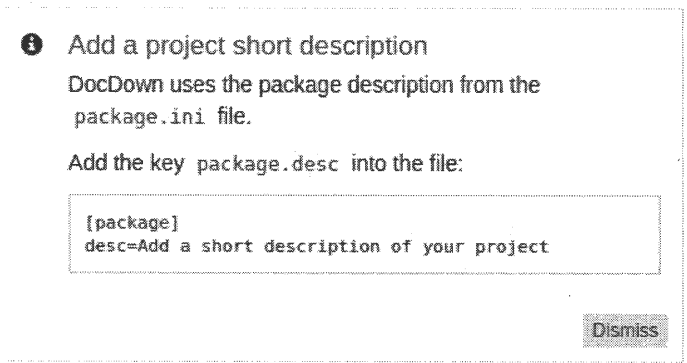


Figure E.1: Exemple de carte suggérant à l'écrivain d'ajouter la clé `package.desc` dans le fichier de package (`package.ini`) de son projet.

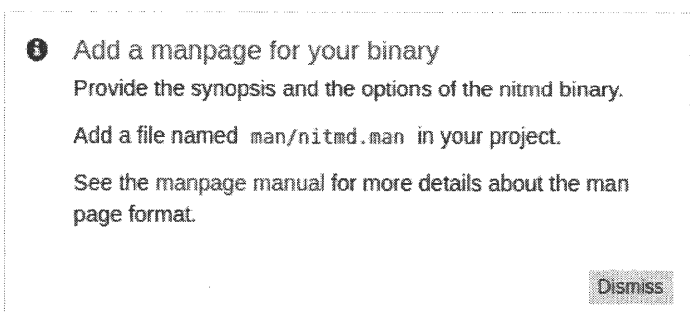
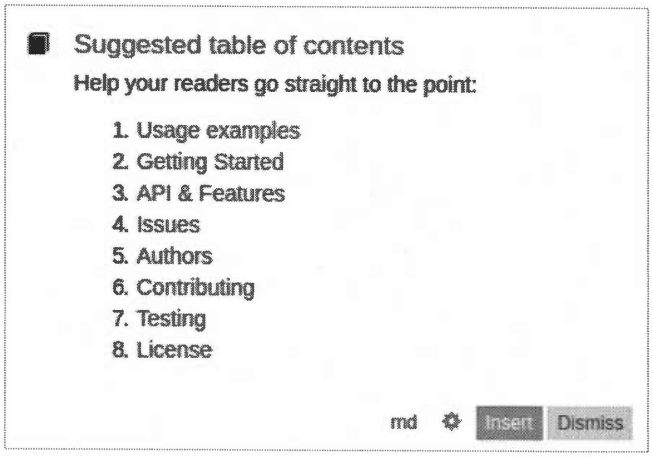


Figure E.2: Exemple de carte suggérant à l'écrivain d'ajouter un fichier `man` pour l'exécutable `nitmd`.

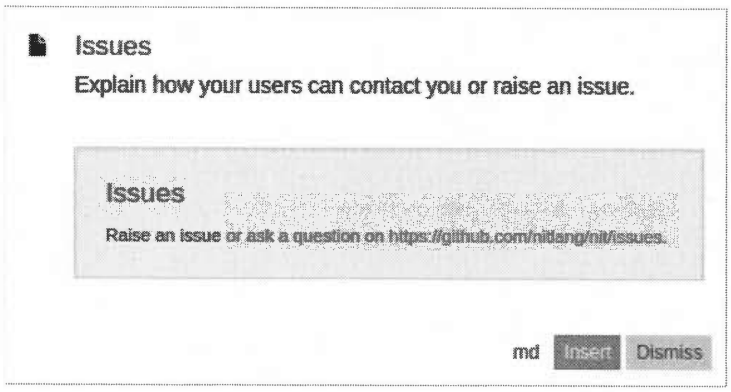


(a) Carte de suggestion.

```
[[toc: markdown]]
```

(b) Contenu Markdown inséré.

Figure E.3: Exemple de carte de suggestion de sommaire (*Summary*) pour le projet `markdown`.

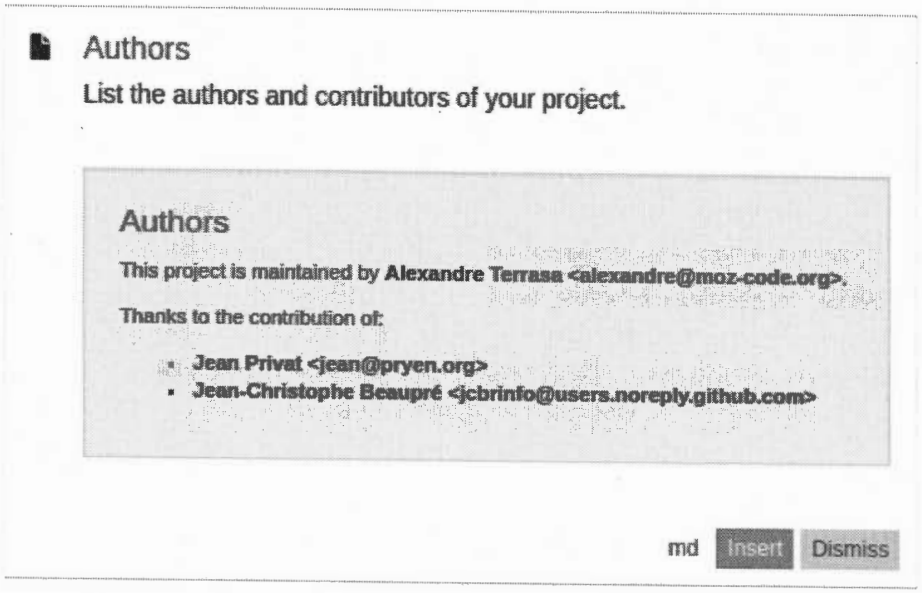


(a) Carte de suggestion.

```
## Issues

Raise an issue or ask a question on
[[ini-issues: markdown]].
```

(b) Contenu Markdown inséré.

Figure E.4: Exemple de carte de problèmes et support (*Issues*) du projet `markdown`.

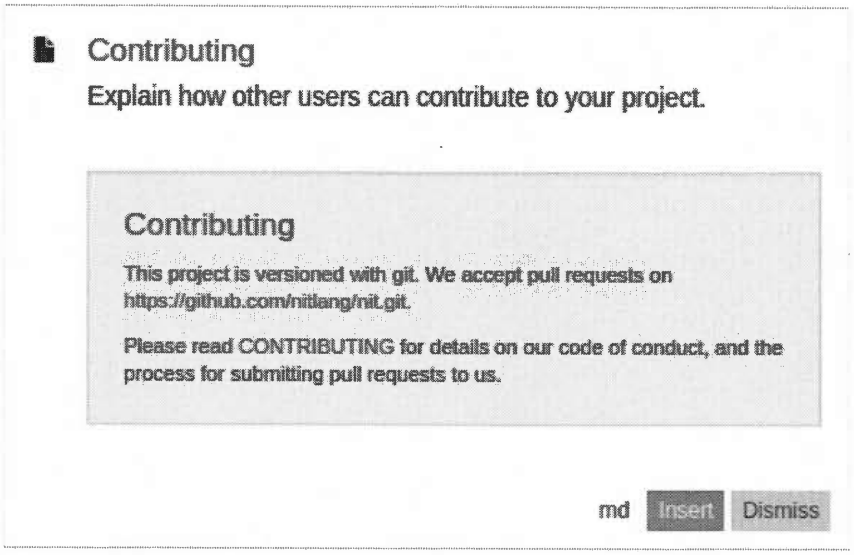


(a) Carte de suggestion.

```
## Authors

This project is maintained by
[[ini-maintainer: markdown]].

Thanks to the contribution of:
[[ini-contributors: markdown]]
```

(b) Contenu Markdown inséré.

Figure E.5: Exemple de carte d'auteurs (*Authors*) du projet `markdown`.

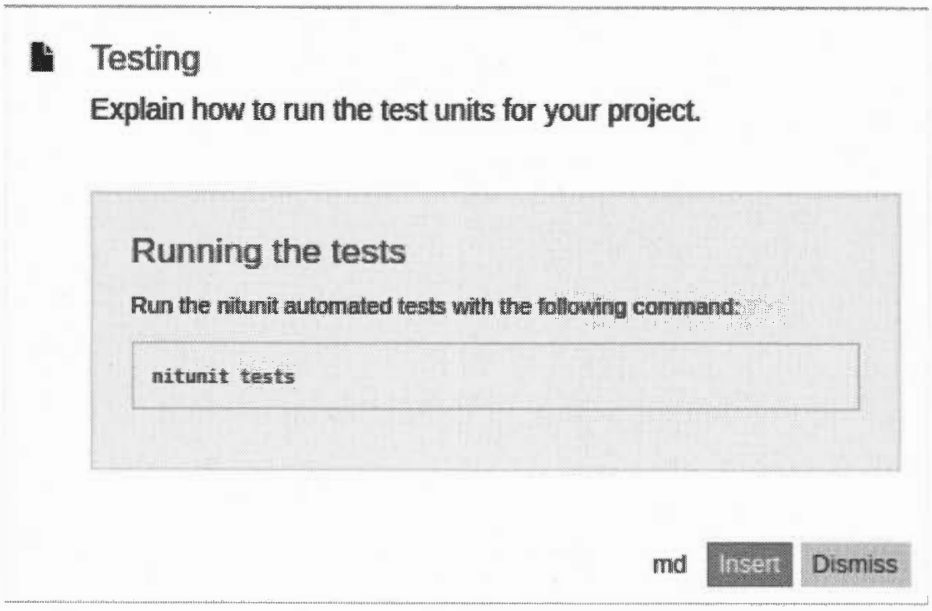


(a) Carte de suggestion.

```
## Contributing

This project is versioned with git.
We accept pull requests on
[[ini-git: markdown]].

Please read [[contrib-file: markdown]] for
details on our code of conduct, and the
process for submitting pull requests to us.
```

(b) Contenu Markdown inséré.

Figure E.6: Exemple de carte de contribution (*Contributing*) pour le projet `markdown`.

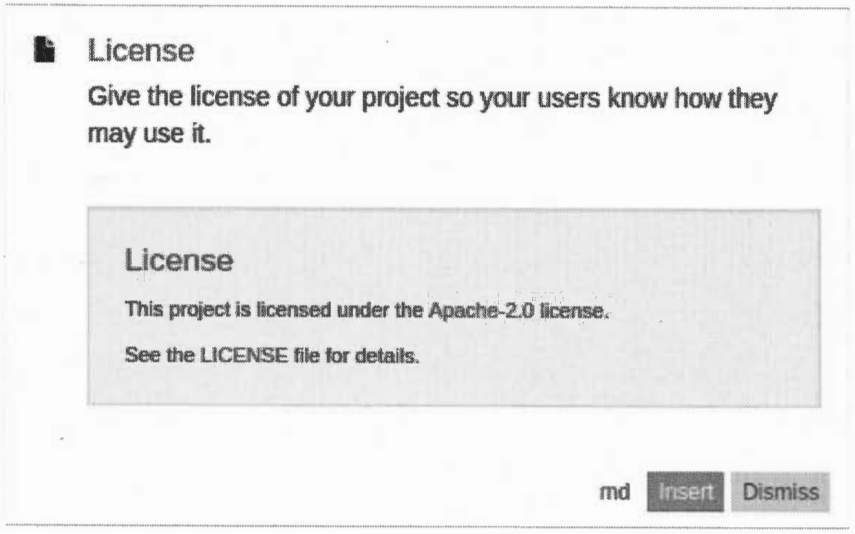


(a) Carte de suggestion.

```
## Running the tests

Run the nitunit automated tests with the following
command:

[[testing: markdown]]
```

(b) Contenu Markdown inséré.

Figure E.7: Exemple de carte de tests (*Testing*) pour le projet `markdown`.



(a) Carte de suggestion.

```
## License

This project is licensed under the
[[ini-license: markdown]] license.

See the [[license-file: markdown]] file for
details.
```

(b) Contenu Markdown inséré.

Figure E.8: Exemple de carte de licence (*License*) du projet `markdown`.

ANNEXE F

PACKAGES NIT UTILISÉS POUR LA VALIDATION DE NITREADME

Tableau F.1: 50 packages Nit dont le fichier README a été généré par `nitreadme` avec l'option `--scaffold`.

array_debug	base64	bcm2835	binary
bitmap	bucketed_game	c	cartesian
cocoa	combinations	config	console
counter	cpp	crapto	crypto
csv	curl	curses	date
deriving	dom	dot	dummy_array
egl	emscripten	event_queue	fca
filter_stream	for_abuse	gen_nit	gettext
glesv2	gmp	graphs	gtk
hash_debug	html	ini	java
jvm	libevent	linux	logic
matrix	md5	meta	mnit
mongodb	opts		

BIBLIOGRAPHIE

18F, General Services Administration (2016a). Making READMEs readable. Récupéré de `https://open-source-guide.18f.gov/making-readmes-readable/`

18F, General Services Administration (2016b). Making repo descriptions short and clear. Récupéré de `https://open-source-guide.18f.gov/writing-the-repo-description/`

Allen, M. (2014). PerlDoc. Récupéré de `http://perldoc.perl.org/`

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A. et Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, *28*(10), 970–983. Récupéré de `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1041053`

Ashkenas, J. (2010). Docco. Récupéré de `http://jashkenas.github.io/docco/`

Bacchelli, A., Lanza, M. et Robbes, R. (2010). Linking e-mails and source code artifacts. Dans *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 375–384. ACM.

Bajolet, L. (2016). Des chaînes de caractères efficaces et résistantes au passage à l'échelle, une proposition de modélisation pour les langages de programmation à objets.

Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P. et Lopes, C. (2006). Sourcerer : A search engine for open source code supporting structure-based search. Dans *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 681–682. ACM.

Beck, K. (2003). *Test-Driven Development—By Example*. Addison-Wesley Professional.

Berglund, E. (2000). Writing for adaptable documentation. Dans *Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation : technology & teamwork (IPCC/SIGDOC'00)*, 497–508. EEE Educational Activities Department Piscataway, NJ, USA.

Brophy, D. (2017). Rebecca. Récupéré de `https://github.com/dave/rebecca`

Brown, D. (2014). PHP.net. Récupéré de `http://php.net/manual`

Buse, R. P. L. et Weimer, W. (2012). Synthesizing API usage examples. Dans *International Conference on Software Engineering (ICSE'12)*, 782–792., Zurich, Switzerland. IEEE Press.

Buse, R. P. L. et Weimer, W. R. (2008). Automatic documentation inference for exceptions. Dans *International Symposium on Software Testing and Analysis*, 273–282., New York, New York, USA. ACM Press. Récupéré de `http://portal.acm.org/citation.cfm?doid=1390630.1390664`

Cao, Y., Zou, Y., Luo, Y., Xie, B. et Zhao, J. (2018). Toward accurate link between code and software documentation. *Science China Information Sciences*, *61*, 1–15.

Chilton, A. (2017). Go ReadMe. Récupéré de `https://go-readme.golang.nz/`

Cleland-Huang, J., Gotel, O. C., Huffman Hayes, J., Mäder, P. et Zisman, A. (2014). Software traceability : trends and future directions. Dans *Proceedings of the on Future of Software Engineering*, 55–69. ACM.

Dabbish, L., Stuart, C., Tsay, J. et Herbsleb, J. (2012). Social coding in GitHub : Transparency and collaboration in an open software repository. Dans *Proceedings of the ACM 2012 conference on computer supported cooperative work*, 1277–1286. ACM.

Dagenais, B. et Robillard, M. P. (2012). Recovering traceability links between an API and its learning resources. Dans *Proceedings of the 34th International Conference on*

Software Engineering (ICSE '12), 47–57., Zurich, Switzerland. IEEE Press. Récupéré de `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6227207`

de Souza, S. C. B., Anquetil, N. et de Oliveira, K. M. (2005). A study of the documentation essential to software maintenance. Dans *International Conference on Design of Communication Documenting & Designing for Pervasive Information (SIGDOC '05)*, p. 68., New York, New York, USA. ACM Press.

Decan, A., Mens, T., Claes, M. et Grosjean, P. (2016). When GitHub meets cran : An analysis of inter-repository package dependency problems. Dans *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 493–504. IEEE.

Denier, S. et Guéhéneuc, Y.-G. (2008). Mendel : A Model, Metrics, and Rules to Understand Class Hierarchies. Dans *IEEE International Conference on Program Comprehension (ICPC'08)*, 143–152. IEEE.

Deursen, A. V. et Kuipers, T. (1999). Building documentation generators. Dans *IEEE International Conference on Software Maintenance (ICSM '99)*, 40–49. IEEE.

Duala-Ekoko, E. et Robillard, M. P. (2012). Asking and answering questions about unfamiliar APIs : An exploratory study. Dans *International Conference on Software Engineering (ICSE '12)*, 266–276., Zurich, Switzerland. IEEE Press.

Dubochet, G. (2011). Scaladoc. Récupéré de `https://wiki.scala-lang.org/display/SW/Scaladoc`

Ducasse, S. et Lanza, M. (2005). The class blueprint : Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, *31*(1), 75–90.

Ducournau, R., Morandat, F. et Privat, J. (2008). *Modules and Class Refinement—A Meta-modeling Approach to Object-Oriented Languages.* Rapport technique, LIRMM - CNRS and Université Montpellier II, Montpellier, France.

Ducournau, R., Morandat, F. et Privat, J. (2009). Empirical assessment of object-oriented implementations with multiple inheritance and static typing. Dans *ACM SIGPLAN Notices*, volume 44, 41–60. ACM.

Dyrynda, M. (2016). Readme Generator. Récupéré de `https://michaeldyrynda.github.io/readme-generator/`

Eriksson, H., Berglund, E. et Nevalainen, P. (2002). Using knowledge engineering support for a java documentation viewer. Dans *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, 57–64. ACM.

Fagin, R., Kumar, R. et Sivakumar, D. (2003). Comparing top k lists. *SIAM Journal on discrete mathematics*, *17*(1), 134–160.

Faulhaber, T. (2009). Autodoc. Récupéré de `http://tomfaulhaber.github.io/autodoc/`

Flatt, M., Barzilay, E. et Findler, R. B. (2009). Scribble : Closing the book on ad hoc documentation tools. *SIGPLAN Not.*, *44*(9), 109–120. `http://dx.doi.org/10.1145/1631687.1596569`. Récupéré de `http://doi.acm.org/10.1145/1631687.1596569`

Forward, A. et Lethbridge, T. C. (2002). The relevance of software documentation, tools and technologies. Dans *Symposium on Document Engineering (DocEng '02)*, 26–33., New York, New York, USA. ACM Press.

Free Software Foundation, Inc. (2016). Making Releases. Récupéré de `https://www.gnu.org/prep/standards/html_node/Releases.html#Releases`

Friendly, L. (1995). The design of distributed hyperlinked programming documentation. Dans *International Workshop on Hypermedia Design '95*, 151–173.

Gamma, E., Helm, R., Johnson, R. et Vlissides, J. M. (1995). *Design patterns : Elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Gélinas, J.-S., Gagnon, É. M. et Privat, J. (2009). Prévention de déréférencement de références nulles dans un langage à objets. *Langages et Modèles à Objets (LMO'09), L-3*, 5–16.

Gentle Bytes (2009). AppleDoc. Récupéré de `http://gentlebytes.com/appledoc/`

Georg Brandl (2007). Sphinx. Récupéré de `http://www.sphinx-doc.org/`

Github, Inc. (2012a). Basic writing and formatting syntax. Récupéré de `https://help.github.com/articles/basic-writing-and-formatting-syntax/`

Github, Inc. (2012b). Creating and highlighting code blocks. Récupéré de `https://help.github.com/articles/creating-and-highlighting-code-blocks/`

Github, Inc. (2018a). About READMEs. Récupéré de `https://help.github.com/articles/about-readmes/`

Github, Inc. (2018b). Formatting your README. Récupéré de `https://guides.github.com/features/wikis/`

Goodger, D. (2016). Markup Syntax and Parser Component of Docutils. Récupéré de `http://docutils.sourceforge.net/rst.html`

Google (2017). SpreadSheet Explore. Récupéré de `http://support.google.com/docs/answer/6280499`

Greene, G. J. et Fischer, B. (2016). Cvexplorer : Identifying candidate developers by mining and exploring their open source contributions. Dans *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 804–809. ACM.

Gruber, J. (2004). Markdown Syntax. Récupéré de `http://daringfireball.net/projects/markdown/`

Gusfield, D. (1997). *Algorithms on strings, trees and sequences : Computer science and computational biology.* Cambridge University Press.

Guthrie, J. T., Britten, T. et Barker, K. G. (2015). Roles of Document Structure, Cognitive Strategy, and Awareness in Searching for Information. *Reading Research Quarterly*, *26*(3), 300–324.

HaL Computer Systems (1991). DocBook. Récupéré de `http://www.docbook.org/`

Hao, Y., Li, G., Mou, L., Zhang, L. et Jin, Z. (2013). MCT : A tool for commenting programs by multimedia comments. Dans *International Conference on Software Engineering (ICSE '13)*, 1339–1342., San Francisco, CA, USA. IEEE Press.

Hassan, F. et Wang, X. (2017). Mining readme files to support automatic building of Java projects in software repositories : Poster. Dans *Proceedings of the 39th International Conference on Software Engineering Companion*, 277–279. IEEE Press.

Hauff, C. et Gousios, G. (2015). Matching GitHub developer profiles to job advertisements. Dans *Proceedings of the 12th Working Conference on Mining Software Repositories*, 362–366. IEEE Press.

Heijstek, W., Kuhne, T. et Chaudron, M. R. (2011). Experimental analysis of textual and graphical representations for software architecture design. Dans *International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*, 167–176. IEEE.

Hens, S., Monperrus, M. et Mezini, M. (2012). Semi-automatically extracting FAQs to improve accessibility of software development knowledge. Dans *International Conference on Software Engineering (ICSE '12)*, 793–803., Zurich, Switzerland. IEEE Press.

Hoffman, D. et Strooper, P. (2001). *API Documentation with Executable Examples.* Rapport technique, University of Queensland.

Holmes, R., Walker, R. J., Murphy, G. C. et Society, I. C. (2006). Approximate structural context matching : An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, *32*(12), 952–970.

Howlett, L. (2013). grunt-readme-generator. Récupéré de `https://www.npmjs.com/package/grunt-readme-generator`

Ikeda, S., Ihara, A., Kula, R. G. et Matsumoto, K. (2018). An empirical study on readme contents for JavaScript packages. *arXiv preprint arXiv :1802.08391.*

JSDoc 3 documentation project (2011). JSDoc. Récupéré de `http://usejsdoc.org/`

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. et Damian, D. (2014). The promises and perils of mining GitHub. Dans *Proceedings of the 11th working conference on mining software repositories*, 92–101. ACM.

Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, *30*(1/2), 81–93.

Kim, J., Lee, S., Hwang, S.-W. et Kim, S. (2013). Enriching documents with examples : A corpus mining approach. *ACM Transactions on Information Systems (TOIS)*, *31*(1), 1.

Klare, G. R. (2000). Readable computer documentation. *ACM Journal of Computer Documentation*, *24*(3), 148–168.

Knuth, D. (1992). *Literate Programming.* Stanford University Center for the Study of Language and Information.

Laferrière, A. (2018). *Applications mobiles portables et de haute qualité : du prototype à la ligne de produits par le raffinement de classes et la programmation polyglotte.* (Thèse de doctorat). Université du Québec à Montréal.

Lethbridge, T. C., Singer, J. et Forward, A. (2003). How software engineers use documentation : The state of the practice. *IEEE Software*, *20*(6), 35–39. Récupéré de `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1241364`

Long, F., Wang, X. et Cai, Y. (2009). Api hyperlinking via structural overlap. Dans *Proceedings of the the 7th joint meeting of the European software engineering conference*

and the ACM SIGSOFT symposium on The foundations of software engineering, 203–212. ACM.

Lutters, W. G. et Seaman, C. B. (2007). Revealing actual documentation usage in software maintenance through war stories. *Information and Software Technology*, *49*(6), 576–587.

MacFarlane, J. (2006). About pandoc. Récupéré de `https://pandoc.org/`

MacFarlane, J., Greenspan, D., Marti, V., Williams, N., Dumke-von der Ehe, B. et Atwood, J. (2004). Why is CommonMark needed ? Récupéré de `http://commonmark.org/`

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J. et McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. Dans *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics : System Demonstrations*, 55–60.

Marcus, A. et Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. Dans *Proceedings of the 25th International Conference on Software Engineering*, 125–135. IEEE Computer Society.

Marlow, S. (2002). Haddock. Récupéré de `https://www.haskell.org/haddock/`

Microsoft (2004). MSDN. Récupéré de `http://msdn.microsoft.com/library`

Microsoft (2006). SandCastle. Récupéré de `https://archive.codeplex.com/?p=sandcastle`

Nasehi, S. M., Sillito, J., Maurer, F. et Burns, C. (2012). What makes a good code example ? A study of programming Q&A in StackOverflow. *IEEE International Conference on Software Maintenance, ICSM*, 25–34.

Neate, B., Irwin, W. et Churcher, N. (2006). CodeRank : A new family of software metrics. *Proceedings of the Australian Software Engineering Conference, ASWEC, 2006*, 369–378.

Oney, S. et Brandt, J. (2012). Codelets : Linking interactive documentation and example code in the editor. Dans *Conference on Human Factors in Computing Systems (CHI '12)*, 2697–2706.

Page, L., Brin, S., Motwani, R. et Winograd, T. (1999). *The PageRank Citation Ranking : Bringing Order to the Web.* Rapport technique, Stanford InfoLab.

Parnin, C. et Treude, C. (2011). Measuring API documentation on the web. Dans *International Workshop on Web 2.0 for Software Engineering (Web2SE '11)*, 25–30., New York, New York, USA. ACM Press.

Pizarro, P. R. (2015). verb. Récupéré de `https://github.com/verbose/verb`

Pizarro, P. R. (2017). README.md Generator. Récupéré de `https://github.com/ppizarror/readme-generator`

Prana, G. A. A., Treude, C., Thung, F., Atapattu, T. et Lo, D. (2018). Categorizing the content of GitHub readme files. *arXiv preprint arXiv :1802.06997.*

Privat, J. et Ducournau, R. (2005). Raffinement de classes dans les langages à objets statiquement typés. *Revue des Sciences et Technologies de l'Information-Série L'Objet : logiciel, bases de données, réseaux*, *11*(1-2), 17–32.

Rackham, S. (2002). Text based document generation. Récupéré de `http://asciidoc.org/`

Reeves, J. (2014). Codox. Récupéré de `https://github.com/weavejester/codox`

Ribreau, F.-G. (2013). Doxx. Récupéré de `https://github.com/FGRibreau/doxx`

Rigby, P. C. et Robillard, M. P. (2013). Discovering essential code elements in informal documentation. *2013 35th International Conference on Software Engineering (ICSE)*, 832–841.

Robillard, M. P. (2009). What makes APIs hard to learn ? Answers from developers. *IEEE Software*, *26*(6), 27–34.

Robillard, M. P. et DeLine, R. (2010). A field study of API learning obstacles. *Empirical Software Engineering, 16*(6), 703–732.

Robillard, M. P., Marcus, A., Treude, C., Bavota, G., Chaparro, O., Ernst, N., Gerosa, M. A., Godfrey, M., Lanza, M., Linares-Vásquez, M. *et al.* (2017). On-demand developer documentation. Dans *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, 479–483. IEEE.

Roehm, T., Tiarks, R., Koschke, R. et Maalej, W. (2012). How do professional developers comprehend software ? Dans *International Conference on Software Engineering (ICSE '12)*, 255–265., Zurich, Switzerland. IEEE Press.

Sadoun, D., Mkhitaryan, S., Nouvel, D. et Valette, M. (2016). Readme generation from an owl ontology describing nlp tools. Dans *Natural Language Generation and the Semantic Web*, 46–49.

Salton, G., Wong, A. et Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM, 18*(11), 613–620.

Schlinkert, J. (2016). generate-readme. Récupéré de `https://www.npmjs.com/package/generate-readme`

Segal, L. (2007). YARD. Récupéré de `http://yardoc.org/`

Sharma, A., Thung, F., Kochhar, P. S., Sulistya, A. et Lo, D. (2017). Cataloging GitHub repositories. Dans *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 314–319. ACM.

Sindhgatta, R. (2006). Using an information retrieval system to retrieve source code samples. Dans *Proceedings of the 28th international conference on Software engineering*, 905–908. ACM.

Subramanian, S., Inozemtseva, L. et Holmes, R. (2014). Live API documentation. Dans *Proceedings of the 36th International Conference on Software Engineering*, 643–652. ACM.

Terrasa, A. et Privat, J. (2013). Efficiency of subtype test in object oriented languages with generics. Dans *Proceedings of the 8th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 4–10. ACM.

Terrasa, A., Privat, J. et Tremblay, G. (2018). Using natural language processing for documentation assist. Dans *AAAI Technical Report WS-18-14, The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence*, 787–790. The AAAI Press. Récupéré de `https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16821`

The Apache Software Foundation (2002). Introduction to the Standard Directory Layout. Récupéré de `https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html`

The Free Software Foundation, Inc. (2005). INI formats. Récupéré de `http://www.nongnu.org/chmspec/latest/INI.html`

The Go Authors (2009). Godoc. Récupéré de `http://blog.golang.org/godoc-documenting-go-code`

Thomas, D. (2001). RDoc. Récupéré de `http://docs.seattlerb.org/rdoc/`

Thung, F., Bissyande, T. F., Lo, D. et Jiang, L. (2013). Network structure of social coding in GitHub. Dans *Software maintenance and reengineering (CSMR), 2013 17th european conference on*, 323–326. IEEE.

Torgersen, M. (1998). Virtual types are statically safe. Dans *5th Workshop on Foundations of Object-Oriented Languages*, 1–9., San Diego, California, USA. K. Bruce.

Treude, C. et Robillard, M. P. (2016). Augmenting API documentation with insights from stack overflow. Dans *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, 392–403. IEEE.

Tsuchiya, R., Washizaki, H., Fukazawa, Y., Oshima, K. et Mibe, R. (2015). Interactive recovery of requirements traceability links using user feedback and configuration

management logs. Dans *International Conference on Advanced Information Systems Engineering*, 247–262. Springer.

Uddin, G. et Robillard, M. P. (2017). Resolving API mentions in informal documents. *arXiv preprint arXiv :1709.02396.*

Universal Software (2001). Universal Report. Récupéré de `http://www.omegacomputer.com/`

Valure, G. (2003). Natural Docs. Récupéré de `http://www.naturaldocs.org/`

van Heesch, D. (2008). Doxygen : Source code documentation generator tool. Récupéré de `http://www.stack.nl/~dimitri/doxygen/`

van Riel, M. (2010). phpDocumentor. Récupéré de `http://www.phpdoc.org/`

Vestdam, T. et Nørmark, K. (2004). Maintaining program understanding—Issues, tools, and future directions. *Nordic Journal of Computing*, *11*(3), 303–320.

Vinárek, J., Hnětynka, P., Šimko, V. et Kroha, P. (2014). Recovering traceability links between code and specification through domain model extraction. Dans *Workshop on Enterprise and Organizational Modeling and Simulation*, 187–201. Springer.

Votruba, T., Hanslík, J. et Nešpor, O. (2010). ApiGen. Récupéré de `http://apigen.org/`

Wang, W. et Godfrey, M. W. (2013). Detecting API usage obstacles : A study of iOS and Android developer questions. Dans *Working Conference on Mining Software Repositories (MSR '13)*, 61–64., San Francisco, CA, USA. IEEE Press.

Wingkvist, A. et Ericsson, M. (2010). A metrics-based approach to technical documentation quality. Dans *International Conference on the Quality of Information and Communications Technology (QUATIC '10)*, 476–481. IEEE.

Yahoo! Inc. (2011). YUIDoc. Récupéré de `http://yui.github.io/yuidoc/`

Ye, D., Xing, Z., Foo, C. Y., Li, J. et Kapre, N. (2016). Learning to extract API mentions from informal natural language discussions. Dans *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, 389–399. IEEE.

Zhang, K. et Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, *18*(6), 1245–1262.

Zhang, Q., Zheng, W. et Lyu, M. (2011). Flow-augmented call graph : A new foundation for taming API complexity. *Lecture Notes in Computer Science*, *6603*, 386–400.

Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q. et Sun, J. (2017). Detecting similar repositories on GitHub. Dans *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, 13–23. IEEE.