Kévin DESPOULAINS
Gaël GENDRON
Corentin GUILLOUX
Valentin FOUCHER
Enzo CRANCE
Charlotte RICHARD
Laure DU MESNILDOT
Timothée NEITTHOFFER
Élèves ingénieurs de l'INSA Rennes
Année universitaire 2018/2019

Rapport final
Projet 4INFO

Logiciel de génération de données d'apprentissage pour la reconnaissance d'écriture manuscrite

Encadrants

Bertrand COÜASNON (*IRISA*)

Erwan FOUCHÉ & Julien BOUVET (*Sopra Steria*)

Projet en collaboration avec

Jean-Yves LE CLERC

(*Archives départementales d'Ille-et-Vilaine*)

Sophie TARDIVEL (*Doptim*)

Table des matières

Chapitre 1

Introduction

Ce projet nous a été proposé par l'équipe IntuiDoc de l'IRISA, en collaboration avec la startup Doptim et avec le soutien de Jean-Yves LE CLERC, conservateur du patrimoine aux archives départementales d'Ille-et-Vilaine. Tout au long de l'année, nous avons été encadrés par Bertrand COÜASNON, enseignant-chercheur membre d'IntuiDoc, Erwan FOUCHÉ, chef de projet chez Sopra Steria et Julien BOUVET, ingénieur chez Sopra Steria également. Nous avons aussi bénéficié des conseils de Sophie TARDIVEL, responsable et *data scientist* chez Doptim.

Ce rapport vient conclure le travail de cette année en faisant le bilan des tâches effectuées. Comme demandé, il prend la forme d'une liste d'annexes expliquant chacune un élément distinct de ce second semestre sur le projet. Nous reviendrons donc d'abord sur les phases de test avant de détailler les modifications apportées aux spécifications au cours du second semestre. Enfin, nous décrirons l'état de finalisation du projet, soit les différences entre le projet final et le projet théorique au moment de la conception, puis nous dresserons le bilan de la planification.

Chapitre 2

Compte rendu des phases de test

Nous avons réalisé des tests successifs pour chaque composant de l'application : la base de données, la découpe d'images, l'API REST, et l'interface. Tous ces tests nous ont permis de nous rendre compte de certaines erreurs qui avaient été commises et de les corriger. Les tests ont été réalisés au fur et à mesure de l'implémentation pour éviter d'accumuler des erreurs que nous n'aurions pas détectées aussi facilement en testant le logiciel une fois totalement construit d'un bout à l'autre. Nous avons effectué ces tests sur chaque brique de notre application ainsi que des tests d'intégration afin de vérifier le bon fonctionnement des connections entre les composants.

2.1 Base de données

En ce qui concerne la base de données, nous avons utilisé des tests unitaires en Scala (notamment grâce à la bibliothèque ScalaTest) afin de tester les fonctionnalités de la base. Nous avons testé toutes les différentes opérations possibles, qui permettent de stocker des données de types différents, de les modifier, et de les supprimer. De plus, nous avons vérifier que l'utilisation de ces fonctions par le reste de l'application ne pose pas de problème.

2.2 Découpe d'images

Pour le traitement d'images, le problème est qu'aucun test non humain ne permet de déterminer si la découpe a été bien réalisée. Nous n'avons donc pas eu d'autre choix que de vérifier manuellement les images obtenues pour vérifier le bon fonctionnement de la découpe. Nous avons sélectionné au hasard un certain nombre d'images parmi celles qui ont été produites pour vérifier le découpage. De plus, divers scripts ont dû être réalisés pour appuyer l'application serveur. Chaque script a été testé lors de son développement, pour s'assurer qu'il fonctionnait correctement, puis en l'utilisant au sein de l'application.

2.3 API REST

Afin de tester l'API REST, nous avons utilisé Postman, un logiciel particulier dédié au test d'API à partir duquel nous avons pu vérifier le bon fonctionnement de l'envoi de nos requêtes ainsi que les réponses obtenues. Postman permet de décrire l'entièreté des requêtes, comme les header, la méthode, le corps de celle-ci. De plus, il permet de vérifier qu'une réponse obtenue est de la forme voulue et ce de manière automatique. On peut également lui demander de lancer un certains nombre de requêtes dans un certain ordre afin de vérifier le comportement général.

2.4 IHM

Enfin, pour tester l'interface web, nous avons effectué une série de test utilisateurs. Pour cela, nous avons essayé chaque fonctionnalité de manière indépendante, puis selon un parcours utilisateur classique avec toutes ses possibilités en vérifiant les résultats à la foi sur le rendu visuel et sur la console intégrée du navigateur web.

Chapitre 3

Mise à jour du rapport de spécification

Nous avons déterminé les spécifications en novembre et la conception en février. Au fur et à mesure de l'évolution du projet pendant le second semestre, nous y avons apporté des mises à jour que nous allons détailler ci-dessous. Nous avons repris la liste des spécifications du rapport de conception, les lignes en bleu sont les règles que nous avons rajoutées pour corriger un cahier des charges incomplet pour le besoin du client, et les lignes en rouge sont celles que nous avons dû écarter par manque de temps. Les deux fonctionnalités principales que nous n'avons pas pu réaliser sont le lien avec un reconnaisseur d'écriture manuscrite et la page de l'interface dédiée à la découpe manuelle des zones du manuscrit.

Bloc 1 : Préparation des données	
Spécification	**Description**
PR_FO_1	Traiter le format PiFF en interne dans le logiciel
PR_FO_2	Fournir un convertisseur du format GEDI vers PiFF
PR_TR_1	Intégrer une fonction de détection de lignes au logiciel
PR_TR_2	Permettre un découpage des images en lignes
PR_TR_3	Localiser les paragraphes
PR_TR_4	Permettre un découpage des images en paragraphes
PR_RE_1	Associer les images à leur transcription
PR_RE_2	Associer la vérité terrain à une transcription
PR_RE_3	Permettre de générer une vérité terrain si besoin, grâce à un reconnaisseur

Bloc 2 : Stockage des données	
Spécification	**Description**
STO_VER	Stocker des imagettes associées à une vérité terrain
STO_USR	Stocker des imagettes associées à une transcription générée par l'utilisateur
STO_REC	Stocker des imagettes associées à une transcription générée par un reconnaisseur
STO_SEL	Fournir des méthodes pour accéder aux données stockées
STO_UPD	Fournir des méthodes pour modifier les données stockées
STO_INS	Fournir des méthodes pour pouvoir insérer des données à stocker
STO_DEL	Fournir des méthodes pour pouvoir supprimer des données stockées

Bloc 3 : Interface avec le reconnaisseur	
Spécification	**Description**
IR_CV	Convertir les données au format d'entrée du reconnaisseur
IR_AP	Fournir les données au reconnaisseur
IR_EV	Pouvoir lancer une évaluation du reconnaisseur
IR_TR	Pouvoir lancer une transcription d'un document par le reconnaisseur

Bloc 4 : Interface avec l'utilisateur	
Spécification	**Description**
PEA_GEN_1	Valider un ensemble d'annotations
PEA_GEN_2	Éditer manuellement les transcriptions
PEA_GEN_3	Corriger les annotations proposées par un reconnaisseur externe à l'application
PEA_GEN_4	Envoyer les modifications à la base de données lorsque la vérité-terrain d'une imagette est modifiée
PEA_GEN_5	Ignorer un couple imagette-transcription s'il n'est pas pertinent
PEA_GEN_6	Regrouper les documents en projets
PEA_GEN_7	Sélectionner d'abord le projet puis le document sur lequel l'utilisateur veut travailler à l'ouverture de l'application
PEA_GEN_8	Créer un nouveau projet
PEA_GEN_9	Basculer vers la page de découpe des zones
PEA_GEN_10	Basculer vers la page d'édition des annotations
PEA_GEN_11	Basculer vers la page de validation des transcriptions
PEA_GEN_12	Supprimer un projet ou un document d'un projet
PEA_GEN_13	Exporter les données d'un projet dans le format d'entrée du reconnaisseur associé au projet
PDEC_OD_1	Créer une nouvelle zone à l'aide d'un rectangle (outil "nouvelle sélection")
PDEC_OD_2	Pouvoir modifier la position des sommets des rectangles
PDEC_OD_3	Rajouter des sommets à la zone
PDEC_OD_4	Changer le type de la zone avec un menu déroulant
PDEC_OD_5	Déplacer la zone sélectionnée sur le document (outil "déplacer")
PDEC_OD_6	Zoomer et dézoomer sur le document (outils "zoom +" et "zoom -")
PDEC_OD_7	Annuler la dernière action (outil "annuler")
PDEC_OD_8	Refaire l'action annulée précédemment (outil "refaire")
PDEC_OD_9	Supprimer toutes les zones de la page pour retourner au document vierge (outil "réinitialiser")
PDEC_OD_10	Appliquer un détecteur de lignes sur la zone sélectionnée (outil "appliquer la détection de lignes sur la zone")
PDEC_OD_11	Continuer la découpe du document sur la page suivante
PDEC_OD_12	Passer à l'édition des annotations sur la page qui vient d'être découpée
PDEC_OD_13	Exporter la page découpée au format PiFF afin de soumettre les données à un reconnaisseur externe à l'application
PDEC_OD_14	Posséder un bouton de retour au menu principal
PDEC_OD_15	Permettre à l'utilisateur de se déplacer sur le manuscrit à l'aide d'un scroll horizontal et vertical
PEMA_1	Placer le curseur sur la première imagette ne possédant pas de transcription
PEMA_2	Positionner le curseur sur l'annotation suivante en appuyant sur Entrée
PEMA_3	Proposer un raccourci clavier permettant de basculer vers la prochaine imagette sans vérité-terrain
PEMA_4	Posséder un bouton intitulé "modifier les zones du manuscrit"
PEMA_5	Afficher la liste des imagettes du document découpé
PEMA_6	Ignorer un couple imagette-transcription s'il n'est pas pertinent
PEMA_7	Basculer vers la page de validation des transcriptions
PCORIA_1	Valider les transcriptions zone par zone et passer à la zone suivante avec un simple appui sur Entrée
PCORIA_2	Pouvoir modifier une annotation fausse en cliquant dessus pour y positionner son curseur et en effectuant ses modifications manuellement
PCORIA_3	Présenter la zone de visualisation des imagettes de la même manière que sur la page d'édition manuelle des transcriptions
PCORIA_4	Posséder également la fonctionnalité de mise à l'écart d'un couple imagette-transcription
PCORIA_5	Basculer vers la page de validation des transcriptions

Spécification	Description
PVAL_1	Accéder à la page de validation depuis le menu principal
PVAL_2	Accéder à cette page de validation depuis les pages d'édition manuelle des annotations et de correction des transcriptions proposées par le reconnaisseur
PVAL_3	Valider les transcriptions zone par zone et passer à la zone suivante avec un simple appui sur Entrée
PVAL_4	Pouvoir modifier une annotation fausse en cliquant dessus pour y positionner son curseur et en effectuant ses modifications manuellement
PVAL_5	Indiquer si les transcriptions ont été fournies manuellement par un humain ou si elles proviennent d'un reconnaisseur
PVAL_6	Faire figurer une fenêtre montrant la page entière découpée en zones avec la zone courante dans une couleur différente
PVAL_7	Valider le travail pour de bon et fermer le document à l'aide d'un bouton prévu à cet effet

Bloc 5 : Lien entre les blocs précédents	
Spécification	**Description**
LINK_PR_STO	Envoyer les données en entrée vers le système de stockage
LINK_STO_IHM	Extraire les données pour les fournir à l'IHM
LINK_STO_IR	Extraire les données pour les fournir au système de reconnaissance
LINK_IHM_STO	Envoyer les demandes de l'IHM au système de stockage
LINK_IHM_IR	Envoyer les résultats du reconnaisseur vers le système de stockage
LINK_COH	Fournir un logiciel composés de blocs communiquant entre eux de manière fonctionnelle et cohérente

Bloc 6 : Général	
Spécification	**Description**
GEN_ERGO	Ergonomie de l'application
GEN_ERGO	Concevoir un logiciel évolutif
GEN_ERGO	Fournir un logiciel open source

Chapitre 4

État de finalisation du projet

En cette fin de projet, nous sommes en capacité de fournir une application fonctionnelle qui remplit les exigences principales du cahier des charges. Nous expliquerons en détail les différentes fonctionnalités implémentées, pour le serveur puis pour le client. Pour mettre en lumière les différences avec le projet tel que nous l'avions imaginé au moment de la conception, cette partie du rapport peut être considérée comme un second rapport de conception, plus bref cette fois-ci, car les éléments qui sont restés identiques à ce premier rapport de conception ne seront pas détaillés. Au besoin, nous introduirons des éléments techniques, facilement compréhensibles pour un lecteur ayant déjà programmé en Java.

4.1 Architecture générale et technologies utilisées

Concernant les technologies, le rapport de conception comportait déjà tous les éléments utilisés. La seule différence est le reconnaisseur d'écriture manuscrite. En effet, par manque de temps, nous n'avons pas pu mettre en place de connexion effective avec un reconnaisseur d'écriture manuscrite. Le reconnaisseur Laia, mentionné dans le rapport de conception, n'est donc pas présent dans le projet, cette partie étant optionnelle dans le cahier des charges initial du projet, qui consiste à principalement générer les bases d'apprentissage. Pour réaliser ce projet, nous avons utilisé SQLite pour la base de données, OpenCV pour la découpe d'images, Angular 7 pour l'interface client, Grizzly pour le serveur web, ainsi que Jersey pour l'API REST. L'architecture du serveur est sensiblement la même, car nous avons souhaité continuer sur le modèle d'un contrôleur qui orchestre toute l'application, découpée en modules : préparation des données, connexion avec la base de données, interface avec le reconnaisseur.

4.2 Côté serveur

Du côté du serveur, nous avons développé une application principale en Scala, ainsi que quelques scripts en Python pour le prétraitement des données XML et le traitement d'images, que nous présenterons également. La différence principale avec le rapport de conception est la présence de ces scripts. Nous avions en effet prévu dans un premier temps de fournir une application serveur monolithique, développée en un seul langage de bout en bout. L'expérience nous a montré qu'il était plus judicieux d'utiliser des scripts comme outils venant appuyer cette application. Nous détaillerons, paquetage par paquetage, le contenu technique et son utilité pour le projet.

4.2.1 Application en Scala

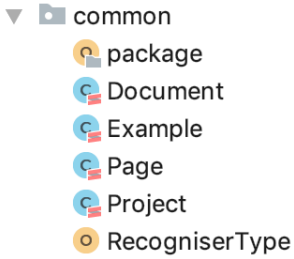
L'application principale se construit autour de deux paquetages en plus de l'objet `Main` qui lance le serveur web :

— un premier paquetage `model` qui contient tous les éléments utiles du serveur, *i.e.* des outils pour traiter les données d'entrée, les images, créer les exemples, et interagir avec la base de données et les reconnaisseurs d'écriture manuscrite ;

— un paquetage `resource` qui contient une classe permettant de recevoir les appels REST du client, d'appeler les bonnes méthodes dans les classes du paquetage précédent, et de façonner les réponses HTTP qui contiennent les résultats de ces méthodes.

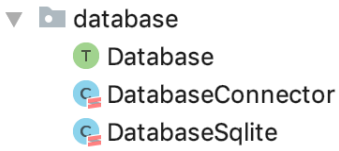
Détaillons à présent le paquetage `model`. Celui-ci est composé de plusieurs sous-paquetages : `common`, `database`, `preparation`, et `recogniser`.

Figure 1 : Paquetage `common`



Commun Le paquetage `common` présenté en Figure 1 contient les classes qui correspondent aux éléments stockés en base de données : des projets contenant des documents composés de pages, ces pages contenant après traitement des exemples d'apprentissage pour les reconnaisseurs d'écriture manuscrite. Le *package object* est une particularité de Scala. Il s'agit d'un objet associé au paquetage, qui contient des définitions utilisées dans le paquetage ainsi qu'en dehors, sémantiquement dépendantes de ce paquetage, mais n'appartenant pas à une classe en particulier. Une valeur `value` pourra être appelée avec `common.value`. Dans ce cas, le *package object* contient des chemins vers différents dossiers ou exécutables sur le système de fichiers, notamment l'endroit où sont stockés les fichiers images ainsi que les exportations d'exemples, ou les scripts de traitement d'images.

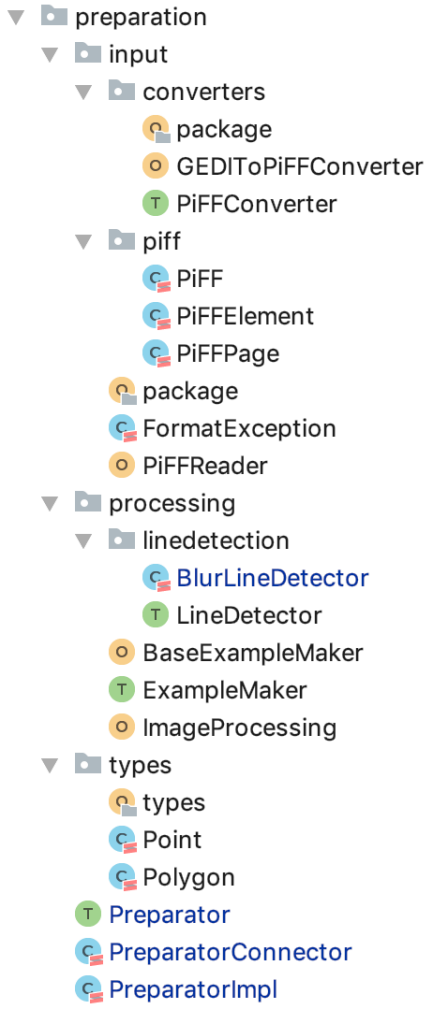
Figure 2 : Paquetage `database`



Base de données Le paquetage `database` contient les classes qui permettent au logiciel de faire des requêtes sur la base de données associée au serveur, qui contient les différents exemples qui seront fournis aux reconnaisseurs d'écriture manuscrite. Le *trait* `Database` (un *trait* en Scala est l'équivalent d'une interface en Java) décrit toutes les actions que l'on doit pouvoir effectuer depuis l'extérieur sur la base de données, et la classe `DatabaseSqlite` l'implémente pour le SGBD SQLite que nous avons choisi pour ce projet. L'utilisation d'un *trait* permet à un développeur réutilisant le projet de changer le SGBD du serveur au besoin. Par exemple, pour un passage réel en production, une base de données MySQL peut être un choix judicieux.

La classe `DatabaseConnector` vient d'un concept que nous avons décidé de suivre tout au long du projet du côté du serveur. Pour chaque paquetage du serveur, le connecteur reçoit les différents appels de l'extérieur, et les relaie vers l'implémentation choisie du *trait* concerné par ce paquetage. Cela permet de choisir localement l'implémentation en changeant une ligne dans un fichier.

Figure 3 : Paquetage `preparation`



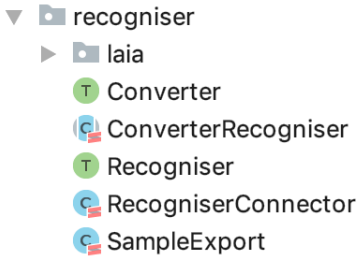
Le paquetage `preparation` est assez complexe. Il est en trois parties : un sous-paquetage `input` qui gère la lecture des données d'entrée, un sous-paquetage `processing` qui réalise le traitement d'images et la création d'exemples, et un sous-paquetage `types` qui contient des définitions d'objets géométriques tels que les points ou les polygones, utiles pour les autres classes du paquetage.

Le sous-paquetage `input` contient la définition du format PiFF, format principal manipulé par notre logiciel. Il contient également un *trait* définissant un convertisseur d'un format quelconque vers PiFF ainsi qu'une implémentation de ce dernier pour traiter le format GEDI dans lequel sont écrites toutes les vérités terrain de la base Maurdor, base d'apprentissage à laquelle nous avons eu accès tout au long du projet. Enfin, il contient un objet `PiFFReader` permettant de lire une vérité terrain dans les formats PiFF (ou GEDI à l'aide des convertisseurs) à partir d'un fichier, et d'en faire un objet `PiFF` manipulable par toutes les autres parties de notre logiciel.

Le sous-paquetage `processing` a quant à lui pour *trait* principal `ExampleMaker`, qui définit un créateur d'exemples à partir d'un objet `PiFF`. Il en contient une implémentation, `BaseExampleMaker`. Ces créateurs d'exemples se basent sur l'objet `ImageProcessing` pour toutes leurs découpes d'images. La découpe est réalisée avec un script Python qui sera présentée plus tard dans ce rapport. Un sous-paquetage `linedetector` est présent. Il s'agit d'un détecteur de lignes fourni par l'encadrant de projet, qui permet de découper les images de manière plus efficace. En effet, la première découpe peut donner des paragraphes, or le reconnaisseur ne s'entraîne que sur des lignes. Il est donc plus pertinent de faire une seconde découpe pour s'assurer que nous obtenons bien des lignes en sortie.

Enfin, ce paquetage `preparation` contient un *trait* `Preparator`, qui définit un objet qui crée des exemples à partir d'une page d'un document et de sa vérité terrain. Comme pour le paquetage précédent, nous fournissons également une implémentation du *trait* ainsi qu'un connecteur.

Figure 4 : Paquetage `recogniser`



Reconnaisseur Dans la première vision de l'interface avec le reconnaisseur que nous avions, nous souhaitions intégrer celui-ci à notre application. C'est-à-dire, pourvoir effectuer depuis celle-ci son entrainement, son évaluation ainsi que son utilisation pour reconnaître des imagettes. Cependant, nous avons dû réduire ces fonctionnalités afin de permettre une meilleure souplesse à l'utilisateur. En effet, celui-ci n'est plus obligé de brancher son reconnaisseur au logiciel afin que celui-ci puisse utiliser la base de connaissance. Cette modification de point de vue nous a également permit de gagner du temps vis-à-vis de cette partie. Ainsi, nous n'avons plus qu'une fonction *export* permettant l'exportation des exemples générés dans le format du reconnaisseur avec les fichiers nécessaires afin d'en faciliter l'utilisation. Nous avons fourni une exportation par défaut qui permet d'obtenir l'image de l'exemple ansi qu'un fichier texte contenant sa transcription. Ces deux fichiers étant nommés de la même manière afin de pouvoir les associer facilement. Cependant, il n'est pas certain que dans le temps restant l'implémentation de l'exportation pour le reconnaisseur de Laia soit opérationnel comme il était prévu à l'origine.

Le paquetage `recogniser` est constitué de deux *trait* qui prennent en charge deux aspect de l'exportation. Le `Converter` permet de contraindre les images de la base au format d'entrée du reconnaisseur selectionné tandis que le `Recogniser` se charge de la création des fichiers nécessaires à l'apprentissage du reconnaisseur à partir de la base. Afin de lier ces deux parties, la classe abstraite `ConverterRecogniser` est utilisée. C'est donc de celle-ci qu'héritent les classes de conversions qui sont à implémenter. Nous fournissons alors les classes `SampleExport` qui est la classe d'export par défaut qui ne formatte donc rien et ne délivre que les images et leur transcription de la facon expliquée précédemment, ainsi qu'une classe `LaiaRecogniser` qui permettra d'exporter pour le reconnaisseur Laia avec les fichiers nécessaires (il n'est cependant pas sur que cette partie soit complètement implémentée dans le temps restant). Ce paquetage possède également un connecteur comme les autres afin de le lier au `Controller`.

Controleur Enfin, ce paquetage `model` contient une classe `Controller` avec toutes les méthodes du modèle utilisées par la ressource REST, ainsi qu'une classe `ImplFactory` donnant toutes les implémentations choisies des différents *traits* dans chaque sous-paquetage.

4.2.2 Scripts Python

Pour appuyer l'application serveur en Scala, nous avons développé trois scripts en Python. Le premier, `tiffsplitter.py`, permet de découper à l'aide d'OpenCV une image TIFF multipages en plusieurs images TIFF avec une seule page chacune. Le second, `gedisplitter.py`, permet de couper un fichier de vérité terrain dans le format GEDI en plusieurs fichiers GEDI, chacun faisant référence à une seule image, donc une seule page. Ce script utilise la bibliothèque defusedxml car l'outil de manipulation des fichiers XML présent dans la bibliothèque standard de Python présente d'après leur site Internet des vulnérabilités logicielles. Ces deux premiers scripts permettent d'avoir des données d'entrée qui correspondent à un couple image - vérité terrain pour une page de texte manuscrit. Le dernier script, `imagecropper`, réalisé aussi avec OpenCV, permet de réaliser les découpes d'images à partir d'une zone rectangulaire donnée en argument.

4.3 Côté client

Notre IHM contient trois pages : une page d'accueil, une page d'annotation manuelle et une page de validation des transcriptions. Nous avions initialement prévu d'inclure également une page de découpe manuelle des zones du manuscrit, mais nous nous sommes trouvés dans l'incapacité de la réaliser par manque de temps.

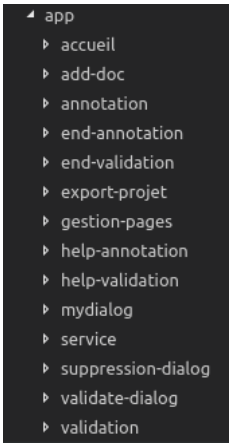
Nous avons réalisé notre interface en `TypeScript`, `Angular` et `HTML`. Elle compte quatorze composants en tout :

— trois composants principaux, soit un pour chaque page de l'interface (accueil, annotation et validation) ;

— dix composants dédiés aux différentes fenêtres de *dialog* qui apparaissent au fil de l'interface. Ce sont des fenêtres qui se superposent à la page de l'interface pour remplir des tâches très précises, comme configurer un nouveau projet, afficher des instructions sur le fonctionnement de l'interface ou encore demander une indication à l'utilisateur ;

— un dernier composant `service` qui se charge d'effectuer certains appels à la ressource REST pour alléger le code des composants des pages principales.

Figure 5 : Arborescence des composants de l'interface



Chaque page de l'interface est dédiée à une fonctionnalité. La page d'accueil permet de naviguer dans les projets et les documents afin de choisir le manuscrit à annoter. Elle propose également de créer de nouveaux projets en choisissant le nom du projet, le reconnaisseur associé, les documents qui le composent et les pages des documents. L'utilisateur peut également supprimer des documents faisant partie d'un projet ou encore des projets entiers.

Sur la page d'annotation manuelle, l'utilisateur peut visualiser le manuscrit découpé ligne par ligne ainsi que la transcription associée sous chaque imagette. Il peut modifier cette transcription en la tapant au clavier. Si un exemple (un couple imagette-transcription) semble manquer de pertinence, l'utilisateur peut le cacher à l'aide d'une croix cliquable située dans le coin supérieur droit de l'imagette. L'exemple est alors grisé sur l'interface et il n'est pas pris en compte lors de l'export des données vers le reconnaisseur du projet. L'utilisateur a encore à tout moment la possibilité de réhabiliter l'exemple en cliquant sur la flèche qui a remplacé la croix en haut à droite de l'imagette.

Enfin, la page de validation permet une relecture rapide des transcriptions pour effectuer une validation finale. L'utilisateur peut également cacher un exemple qui manque de pertinence, comme sur la page d'annotation. Un simple appui sur `Entrée` permet de valider d'un seul coup toutes les transcriptions affichées sur la page courante.

Nous avons développé l'ergonomie de notre application autant que possible dans le temps restreint dont nous disposions et nous avons tenté de fournir une application aussi intuitive que possible en facilitant son utilisation et en simplifiant les gestes requis par l'utilisateur (minimum de mouvements de souris, raccourcis clavier, ...). Cependant, l'ergonomie n'est pas optimale et nécessiterait des améliorations, notemment au niveau de la répétition de certains gestes par l'utilisateur qui requièrent de la précision. Toutefois, nous avons préféré mettre l'accent sur la stabilité et le bon fonctionnement de notre application avant de dédier le temps restant à l'amélioration de l'ergonomie.

Chapitre 5

Bilan de planification

Conformément au rapport de conception, nous avons construit le projet sur deux itérations. La première réduisait l'interface au maximum tout en répondant au cahier des charges. Cette interface permettait donc d'ouvrir un document de travail, et de valider ou invalider les transcriptions. La première itération a été rendue avec un mois de retard, en accord avec l'encadrant de projet. Le but est ici d'en expliquer les raisons. Nous nous intéresserons d'abord aux problèmes rencontrés, à la quantification des impacts de ces problèmes, et aux connaissances acquises pour les projets futurs.

5.1 Problèmes rencontrés

Lors de la phase de réalisation du projet au second semestre, nous avons été confrontés à plusieurs problèmes, listés comme risques dans le rapport de planification pour la plupart. Nous allons détailler les trois problèmes principaux.

Le premier élément qui nous a concernés faisait partie de la liste des risques dans le rapport de planification. Il s'agit de la surcharge de travail avec les cours. Pour éviter ce problème, nous avions prévu de prendre de l'avance au semestre précédent, qui était cependant très chargé. Nous avons réussi à dégager des heures de travail, qui ont servi à installer et mettre en place les différentes technologies nécessaires au projet. Cette phase d'installation a pris beaucoup plus de temps que prévu, et nous nous sommes retrouvés avec une installation partiellement effectuée en début de S8, mais très peu de code utile au projet préparé à l'avance. Cette solution d'anticipation n'a donc pas fonctionné, et nous avons pris du retard. Un autre facteur était la quantité d'éléments présents dans le cahier des charges. En effet, nous avons vu trop grand au départ, en sous-estimant le temps nécessaire pour développer le logiciel, ce qui semble toutefois être une erreur récurrente chaque année en 4INFO.

Un second élément inattendu était une mauvaise pratique logicielle. En d'autres termes, dans le souci de bien faire, et de par nos habitudes en cours, nous avons souhaité dans un premier temps réaliser l'intégralité du projet en un seul bloc, tout le code étant écrit en Scala ou Java. Ceci s'est révélé être une solution beaucoup moins efficace que celle que nous avons finalement mise en place, qui consiste à utiliser des scripts externes pour nous aider au besoin. Par exemple, pour traiter les documents de la base Maurdor, nous avons écrit un script Python qui parcourt un dossier et qui découpe les pages des documents ainsi que leurs vérités terrain, pour obtenir un couple image - vérité terrain par page de chaque document. La découpe d'images est également réalisée avec un script externe. La bibliothèque graphique est toujours OpenCV comme prévu initialement, mais cette bibliothèque a un très mauvais support de Java, et est beaucoup plus mature en C++ et en Python. Il était donc plus raisonnable d'utiliser la bibliothèque dans un script Python, et de l'appeler depuis notre application en Scala, d'autant plus que Scala propose une manière concise et élégante d'appeler des commandes externes, via un opérateur dédié à cette tâche.

Le dernier élément auquel nous avons été confrontés est l'absence d'un des membres du groupe, Valentin. En effet, il avait déjà été beaucoup absent au premier semestre pour raison personnelle, et n'avait pu nous aider que sur le rapport de pré-étude du projet. Nous pensions pouvoir le réintégrer à l'équipe au second semestre,

en lui proposant des tâches qui pourraient nous aider sans pour autant prendre trop de risques, c'est-à-dire que nous lui avons confié des tâches non critiques. Il n'a malheureusement pas pu les effectuer, et a quitté l'équipe du projet car il souhaite quitter l'INSA et se réorienter. Ce sont des choses qui arrivent, mais ayant prévu des heures pour Valentin à la fois dans la planification et en réunion avec l'encadrant au second semestre, nous avons réparti son travail entre les membres du groupe, et avons effectué l'intégralité de la partie technique du projet à quatre, ce qui a accentué l'effet du premier élément mentionné, à savoir la surcharge de travail. Au total, un mois de retard a été comptabilisé, et accepté par l'encadrant au vu de ces circonstances particulières.

5.2 Quantification des impacts

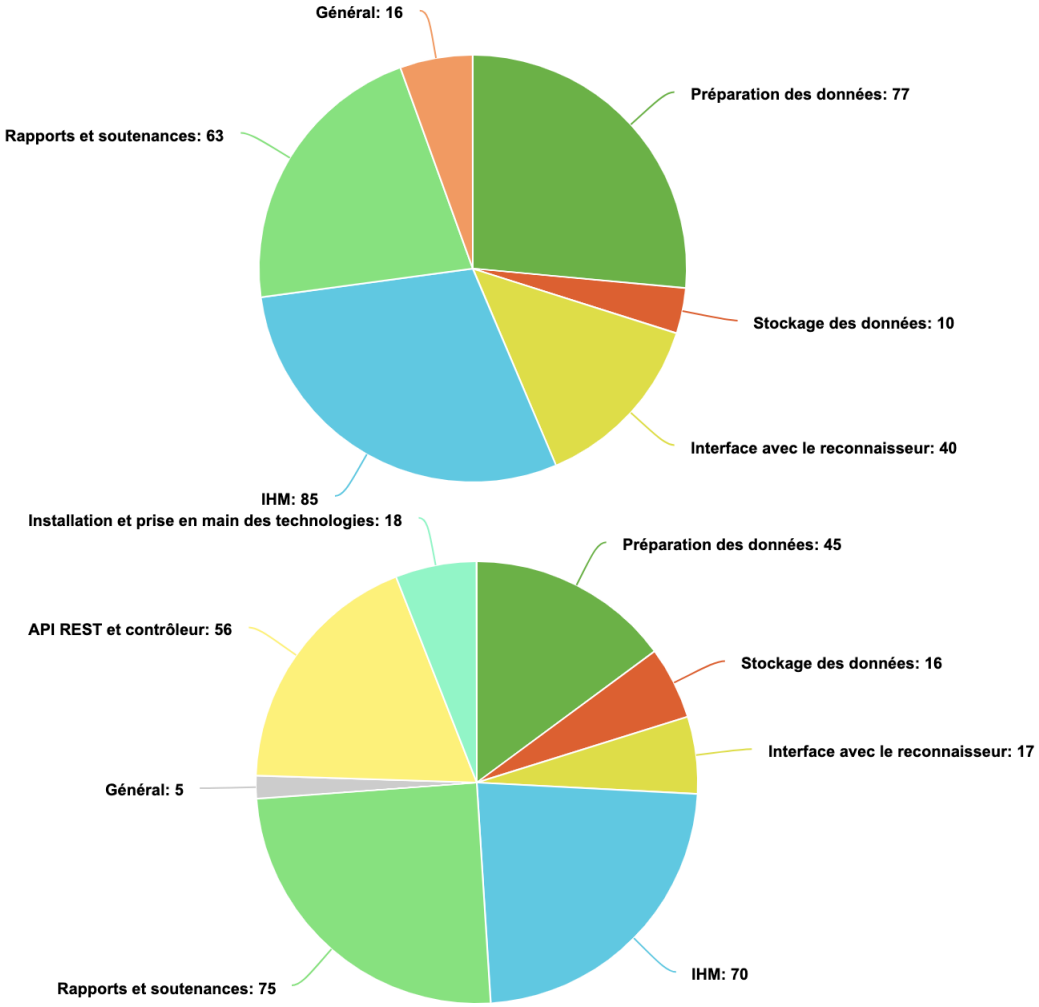
Pour garder une trace du temps passé sur chaque partie du projet, nous avons tenu une table des temps pour chaque personne sur un tableur partagé entre nous. Cet outil nous permet de dresser le tableau comparatif final ci-dessous.

Comparaison des temps prévus et réels		
Sous-partie du projet	**Temps prévu**	**Temps réel**
Préparation des données	77h	45h
Stockage des données	10h	16h
Interface avec le reconnaisseur	40h	17h
Interface Homme-Machine	85h	70h
Rapports et soutenances	63h	75h
Général	16h	$\sim$ 5h
API REST et contrôleur	0h	56h
Installation et prise en main des technologies	0h	18h
Total	**291h**	$\sim$ **302h**

Au vu du rapport de planification et du temps de travail effectué, on peut voir que la différence entre les heures de travail théoriques sur le projet et les heures réelles est faible dans le total, mais conséquente si l'on regarde partie par partie, pour plusieurs raisons. La première raison est que nous avons prévu des marges et évalué toutes les durées à la hausse par sécurité, ce qui augmente le nombre d'heures à passer sur le projet pour être certain de le terminer à temps. Cela explique la différence d'heures pour les parties de préparation des données, interface avec le reconnaisseur, IHM. La deuxième raison est que nous avons passé du temps sur des parties non prévues dans la planification initiale. Par exemple, la définition, l'implémentation, et le test de l'API REST ont pris un nombre d'heures conséquent, qui n'était pas prévu dans le rapport de planification. Il en est de même pour l'installation et la prise en main des technologies. Nos disponibilités par semaine ont donc été une bonne estimation, mais la répartition de ces heures l'était moins, comme nous pouvons le constater dans la Figure 6.

En termes de dates, le projet avait été planifié sur deux itérations, la première étant pour le 27 février et la seconde pour le 26 avril. La première itération a été rendue le 5 avril, et la seconde en fin d'année, le 9 mai. Un mois de retard a donc été comptabilisé pour la première itération, qui était peu solide et allégée par rapport aux éléments prévus. Il était possible de visualiser des exemples déjà ajoutés dans la base de données, ainsi que de créer des nouveaux projets, mais tous les éléments de ces projets devaient être ajoutés à la base de données manuellement, ce qui en fait une version peu utilisable. La seconde itération a rattrapé la moitié de ce retard, du fait d'une simplification du cahier des charges, et d'une mise en oeuvre plus expérimentée au niveau technique. Les fonctionnalités essentielles pour générer des ensembles d'entraînement pour les reconnaisseurs d'écriture manuscrite sont réunies, comme nous pouvons le constater dans une annexe précédente de ce rapport. D'autres fonctionnalités prévues ont été annulées. Ayant un nombre d'heures global cohérent avec la théorie, on a donc ici la confirmation que nous avons sous-estimé le nombre d'heures de travail sur chaque partie.

Figure 6 : Répartitions théorique et réelle des heures du projet



5.3 Conclusions et connaissances acquises pour les projets futurs

Pour résumer, on peut dire que le retard est lié à trois éléments : une mauvaise estimation du temps nécessaire sur les différentes tâches, des mauvais choix logiciels au début du projet, et le départ d'un membre de l'équipe pour la phase de développement.

Le premier point nous a apporté de l'expérience en gestion pour les projets futurs. En effet, nous avons à présent vécu un projet dans lequel nous avons sous-estimé le temps pris par les différentes tâches de la phase de développement. En d'autres termes, nous pouvons dire que nous avons obtenu la preuve empirique que le développement ne peut pas se résumer à l'écriture du code. L'installation et la configuration des différents logiciels et technologies utilisés peut prendre un temps non négligeable, ce à quoi nous n'avons pas fait assez attention. Partir d'une solution simple, puis ajouter peu à peu des améliorations logicielles, c'est-à-dire voir le cahier des charges comme un objet itératif, semble également être une méthode à privilégier.

Le second point nous a apporté de l'expérience quant à lui sur la phase de développement. Nous avons bien compris le fait qu'une application en bloc unique entièrement écrite dans le même langage n'est pas toujours la meilleure solution. Dans un tel projet destiné à des chercheurs, ajouter un langage (qui plus est, répandu) dans les dépendances du logiciel n'est pas une contrainte, et les développeurs ne doivent pas hésiter à scripter certaines tâches, surtout lorsque cette solution est beaucoup plus simple que de s'entêter à vouloir intégrer ces tâches à l'intérieur du logiciel principal. Les langages de script, comme leur nom l'indique, sont faits pour répondre à ce genre de besoins, en permettant de développer des scripts annexes qui viennent appuyer efficacement un plus gros projet.

Enfin, le dernier point a permis de montrer que malgré toute planification préalable, un projet peut toujours avoir des imprévus d'ordre plus ou moins important. Dans notre cas, développer la partie technique d'un projet à quatre et sans reprise de code a été un réel défi, et nous avons fait au mieux lors de ce semestre pour livrer un logiciel fonctionnel en conservant un maximum d'éléments du cahier des charges d'origine.

INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839
35708 Rennes Cedex 7

Tél. +33 (0) 2 23 23 82 00
Fax +33 (0) 2 23 23 83 96

www.insa-rennes.fr

INSA

UNIVERSITE BRETAGNE LOIRE

Cti Commission des Titres d'Ingénieur

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR, DE LA RECHERCHE ET DE L'INNOVATION