Rapport

Technologie Objet & Gestion de Projet
Navigateur Web

Groupe 5

Adrien CAUBEL
Guillaume CHAUDON
Simon JANDA
Simon MOULIN
Victor NANCHE

1ère année par apprentissage
Promotion 2023

INP TOULOUSE ENSEEIHT

Table des matières

1 Introduction

1.1 Le projet

Ce projet s'inscrit dans le cadre des unités d'enseignement Technologie Objet et Gestion de Projet dispensées durant le second semestre du cycle ingénieur. L'objectif principal de ce projet est de mettre en application les principes agiles vus en Gestion de Projet. L'implémentation de l'application quand à elle, est un objectif secondaire car les notions de programmation ont été travaillées en TDs et TPs. Il était donc conseillé d'avoir un projet ambitieux qui serait difficile à concrétiser dans le temps imparti : la création d'un navigateur web.

1.2 Liens utiles

L'avancement et les sources du projet sont disponibles aux liens suivants :

- Github https://github.com/simjnd/navigateur/wiki
- Trello https://trello.com/b/HyinoLEE/navigateur
- SVN http://cregut.svn.enseeiht.fr/2020/1air/cpo/pl/E5

2 Organisation de l'équipe

2.1 Itération initiale

Cette première itération a été consacrée à l'organisation du travail. Après avoir assemblé notre équipe et nous être mis d'accord sur le projet que nous réaliserions ensemble, nous avons jugé important de consulter chaque membre de l'équipe afin de connaître sa vision et ses ambitions pour notre application. Nous avons ainsi pu nous accorder sur une vision commune : pas seulement un navigateur de base, mais un, dont les fonctionnalités rivalisent avec les navigateurs modernes. De cette vision a découlé une idée générale des fonctionnalités principales de notre navigateur.

Avant d'aller plus loin, nous nous sommes mis d'accord sur les outils que nous utiliserions. Nous avons commencé par créer un tableau Kanban sur Trello afin de transcrire notre avancée sur le projet et l'état des users stories. Ensuite, un dépôt Git a été créé sur la plateforme GitHub. Sur ce dépôt nous avons également mis en place un wiki interne sur lequel nous détaillons le workflow Git. Afin de faciliter l'entraide, nous utiliserons l'IDE IntelliJ.

Une fois les outils en place, nous avons pu commencer à fournir le backlog du projet. Nous avons commencé par lister les épiques importantes : la gestion des données de navigation, les onglets, l'historique, le gestionnaire de téléchargement. Nous les avons ensuite décomposées en user stories individuelles avec une évaluation des points d'efforts et de leur valeur métier. Pour les métriques, nous avons respectivement utilisé les tailles de t-shirt (S, L, XL), et une échelle de 1 à 10.

2.2 Organisation générale

Après avoir identifié les différentes épiques et user stories, nous avons procédé à un planning poker pour déterminer les points d'efforts et valeurs métier de ces stories.
Nous avons ensuite fournit le backlog du premier sprint, au début duquel nous avons chacun choisi les stories sur lesquelles nous souhaitions travailler. Au début des périodes de travail nous partagions des croissants pour du teambuilding et nous réalisons un "daily" au cours duquel nous faisions le point sur notre progression, puis au cours des périodes de travail nous communiquions pour être sûr qu'aucun de nous ne soit bloqué.

2.3 Organisation à chaque sprint

Au début de chaque sprint nous menions un *Sprint Planning* au cours duquel nous établissions l'objectif et la vision du sprint dans le but de fédérer et motiver l'équipe. De plus, cela nous permettait d'avoir une vision globale de l'avancemenet du projet.

Cette vision nous guidait dans le choix des *User Stories* à réaliser dans le sprint. Nous cherchions un équilibre dans les valeurs métiers et points d'efforts afin de ne pas avoir une surcharge avec trop d'user stories critiques. Cependant, il nous fallait quand même un backlog de sprint qui correspond à notre vélocité.

Chaque membre de l'équipe choisissait ensuite la prochaine user story sur laquelle il souhaitait travailler. Il créait ainsi une nouvelle une branche de travail sur git. Une fois sa story implémentée et testée, il devait résoudre les conflits avec la branche principale (branche `dev`). Une fois le code poussé sur GitHub une *Pull Request* est créée afin que les autres membres de l'équipe effectuent et une validation.

Au début de chaque séance de travail, nous menions un *Daily* au cours duquel chaque membre faisait un point de son avancement, évoquait ses problèmes bloquants, et ce qu'il souhaitait faire durant la séance de travail.

Au cours d'un sprint nous prenions du temps pour un *Sanity Check* durant lequel nous nous consultions afin de voir si les users stories engagées ne nécessitaient pas un nouveau raffinage, et si les points d'effort avaient été mesurés précisément. Plus particulièrement, au premier sprint nous avons du raffiner des users stories par manque d'expérience.

La fin de chaque sprint donnait lieu à plusieurs échanges : d'abord une *Sprint Review*, au cours de laquelle nous présentions la nouvelle version du navigateur, soit à l'encadrant quand c'était possible, soit entre nous afin que l'équipe puisse voir le résultat des efforts engagés. Ensuite une *Rétrospective de Sprint* au cours de laquelle nous évaluions notre alignement avec nos objectifs : le respect des méthodes agiles, l'ambiance et l'attitude de travail. Ainsi, nous décidions ensemble des mesures à adopter si besoin pour corriger notre trajectoire. Enfin nous faisions une session d'entretien du backlog préalable au sprint planning suivant. Cela nous permettait de réévaluer les points d'efforts, de raffiner des users stories grâce à l'expérience acquise.

3 Découpage des fonctionnalités

Avant de débuter la conception du projet, nous avons dû établir les principales fonctionnalités puis définir leur point d'effort et leur valeur métier. Une fois ces deux métriques définies, nous avons établie les fonctionnalités à faire dans chaque sprint.

3.1 Liste des fonctionnalités

Epiques	Fonctionnalité	Point d'effort	Valeur métier
Fenetre	Avoir l'affichage	M	10
	Bouton rafraîchir	S	8
	Bouton précédent	S	8
	Bouton suivant	S	8
	Barre de recherche	S	10
	Bouton Menu	S	6
Historique	Enregistrer historique	XL	4
	Afficher historique	L	4
	Supprimer historique	S	4
	Rechercher historique	S	3
	Cliquer sur lien dans historique	M	4
Onglet	Ajouter onglet	XL	8
	Naviguer entre onglet	L	8
	Supprimer onglet	M	8
	Interchanger onglet	XL	3
	Grouper onglet	XL	3
Barre recherche	Écrire dans la barre	M	5
	Loader (barre de chargement)	S	3
	Sauvegarder les cookies	L	6
	Suggestion basée sur historique	L	6
	Choisir son moteur de recherche	L	6
Favoris	Ajouter la page actuelle favoris	M	7
	Modifier nom d'un favori	L	5
	Modifier adresse principale favori	L	5
	Créer un dossier	L	7
	Cliquer sur un favori	S	7
	Modifier nom dossier	L	5
Mot de passe	Proposer génération lors d'une inscription	L	4
	Saisie automatique du mot de passe	XL	4
	Afficher liste des mots de passes	M	4
	Détection des mots de passe pwned	M	2
Téléchargement	Choix du dossier destination	M	4
	Etat du téléchargement	XL	5
	Liste des téléchargement	M	4
Données navigations	Sauvegarder cookies	L	6
	Sauvegarde localstorage	L	7
Navigation privée	Navigation privée	M	6

Vous pouvez retrouver ces mêmes user stories sur le Trello

3.2 Fonctionnalités sprint n°1

Ce premier sprint a été consacré à la mise en place de l'infrastructure nécessaire pour l'utilisation d'un navigateur web. Etant donné que nous ne connaissions pas nos capacités, nous avons rajouté de nouvelles tâches lors du *Sanity Check*. On retrouve ainsi les fonctionnalités suivantes.

epiques	Fonctionnalité	Point d'effort	Valeur métier	Affectée à
Fenetre	Avoir l'affichage	M	10	Victor
	Bouton rafraîchir	S	8	Guillaume
	Bouton précédent	S	8	Guillaume
	Bouton suivant	S	8	Guillaume
	Barre de recherche	S	10	Victor
	Bouton Menu	S	6	Simon J
Historique	Enregistrer historique	XL	4	Simon M, Adrien
	Afficher historique	L	4	Simon M, Adrien
Barre recherche	Loader (barre de chargement)	M	5	Simon J

Ainsi, nous avons pu calculer la vélocité de l'équipe. Étant donné l'utilisation de taille de T-shirt, nous avons défini une conversion arbitraire $S = 3, M = 5, L = 8, XL = 13$. Ainsi, la vélocité du premier sprint était de 46.

3.2.1 Conclusion du sprint

Ce premier sprint nous a permis d'avoir une première estimation de la vélocité de l'équipe. De plus, il y a eu une très bonne entente entre les membres de l'équipe, et toutes les tâches fixées ont été réalisées. Cela était donc prometteur pour la suite du projet.

3.3 Fonctionnalités sprint n°2

Durant le deuxième sprint nous nous sommes concentré sur la mise en place d'outils pour faciliter la navigation à l'utilisateur.

Épiques	Fonctionnalité	Point d'effort	Valeur métier	Affecté à
Onglet	Ajouter un onglet	XL	8	Adrien, Guillaume
	Naviguer entre onglet	L	8	Adrien, Guillaume
	Supprimer un onglet	L	8	Adrien, Guillaume
Historique	Rechercher historique	S	3	Simon M, Victor
	Supprimer historique	S	4	Simon M, Victor
Données navigation	Sauvegarder cookies à la fermeture	L	6	Simon J

La vélocité théorique du sprint était de 48. Mais comme expliqué ci-dessous, nous n'avons pas réussir à finir toutes les fonctionnalités.

3.3.1 Conclusion du sprint

Nous avons mal évalué la difficulté pour la mise en place des onglets de navigation. En effet, la tâche s'est avérée plus longue que prévue. De nombreux problèmes auxquels nous n'avions pas pensés nous ont obligé à modifier notre conception et une partie du code réalisé au premier sprint.
Par conséquent, l'épique *onglet* sera continuée sur le troisième sprint.

3.4 Fonctionnalités sprint n°3

Suite à un léger retard pris dans le sprint précédent, nous avons donc du terminer le développement des onglets. Et, le reste de l'équipe a pu se consacrer au développement de nouvelles fonctionnalités.

Epiques	Fonctionnalité	Point d'effort	Valeur métier	Affecté à
Onglet	Ajouter un onglet	XL	8	Guillaume, Adrien
	Naviguer entre onglet	L	8	Guillaume, Simon J
	Supprimer un onglet	L	8	Guillaume, Simon J
Favoris	Créer un dossier	L	7	Simon M
	Ajouter un favori	M	7	Simon M
	Cliquer sur un favori	S	7	Simon M
Barre de recherche	Choisir son navigateur (fichier de configuration)	L	6	Victor
Navigation privée	Navigation privée	M	6	Guillaume
Données de navigation	Sauvegarde *localStorage*	L	7	Simon J
Autre	Revue de code BDD			Adrien
	Revue de code fichier de conf.			Adrien

La fonctionnalité *Choisir son navigateur* a été sous-évaluée. Bien qu'elle soit finie, elle nous a pris plus de temps que prévue.

La vélocité de ce sprint est de 69, ce qui est largement au-dessus des deux dernières vélocités. Mais, il faut considérer que l'épique *onglet* s'est réalisée sur deux sprints. On peut tout de même se demander si certains points d'effort n'ont pas été sur-estimés.

3.4.1 Conclusion du sprint

Le retard a été rattrapé rapidement et nous avons pu développer autant de fonctionnalités que prévu (sans compter les fonctionnalités en retard). Ce troisième et dernier sprint nous a permis d'avoir une application fonctionnelle et mature.

3.5 Fonctionnalités manquantes

Ce dernier tableau résume les fonctionnalités qui n'ont pas pu être développées par manque de temps. Cependant, ces fonctionnalités sont peu valorisantes pour l'application, là, où, nous avons voulu nous concentrer sur les services essentiels. En effet, aucune de ces fonctionnalités n'ont une valeur métier importante et leurs points d'effort sont assez élevés. Ces fonctionnalités n'étaient donc pas prioritaires.

Épiques	Fonctionnalité	Point d'effort	Valeur métier
Favoris	Modifier nom d'un favori	L	5
	Modifier adresse principal du favori	L	5
	Modifier nom d'un dossier	L	5
Mot de passe	Proposer génération lors d'une inscription	L	4
	Saisie automatique du mot de passe	XL	4
	Afficher liste des mots de passes	M	4
	Détection des mots de passe pwned	M	2
Téléchargement	Choix du dossier destination	M	4
	Etat du téléchargement	XL	5
	Liste des téléchargement	M	4

4 Choix des technologies

4.1 Langage et cadriciels

Pour ce projet le langage Java était imposé, mais nous avions le choix pour les bibliothèques et cadriciels utilisables.

Pour l'interface graphique nous avons donc choisi JavaFX pour plusieurs raisons. La principale est la présence des composants `WebEngine` et `WebView` qui nous permettaient de ne pas re-développer un moteur de rendu de pages Web qui aurait rendu notre projet difficilement réalisable en 3 semaines.
D'autres raisons nous ont incité à utiliser JavaFX :

- SceneBuilder qui permet de réaliser des interfaces graphiques plus simplement à partir d'un fichier fxml
- Composants stylisables avec du CSS.
- Meilleure gestion des événements que Swing.

De plus nous avions déjà tous fait du Java Swing à l'IUT mais aussi à l'ENSEEIHT et nous voulions donc essayer une autre technologie.

4.2 Gestionnaire de dépendances

Afin de rendre le développement de notre application plus simple nous avons décidé d'utiliser un gestionnaire de dépendances. Cela rend l'installation de dépendances automatique et commune afin d'éviter qu'un développeur se retrouve avec des problèmes de paquets.
Nous avons donc choisi |Maven| qui est un outil de gestion de dépendances mais aussi d'automatisation pour l'intégration continue. Cela nous a permis d'avoir à configurer seulement un fichier `pom.xml` et Maven s'occupait d'ajouter les paquets dans notre projet.

4.3 Stockage des données

Pour le stockage des données nous avions deux besoins :

- Fichier de configuration : peu de données mais devant être accessible rapidement.
- Données de l'utilisateur : grosse quantité de données.

Pour les fichiers de configuration nous avons choisi d'utiliser le format `JSON` qui est simple à créer et à utiliser.

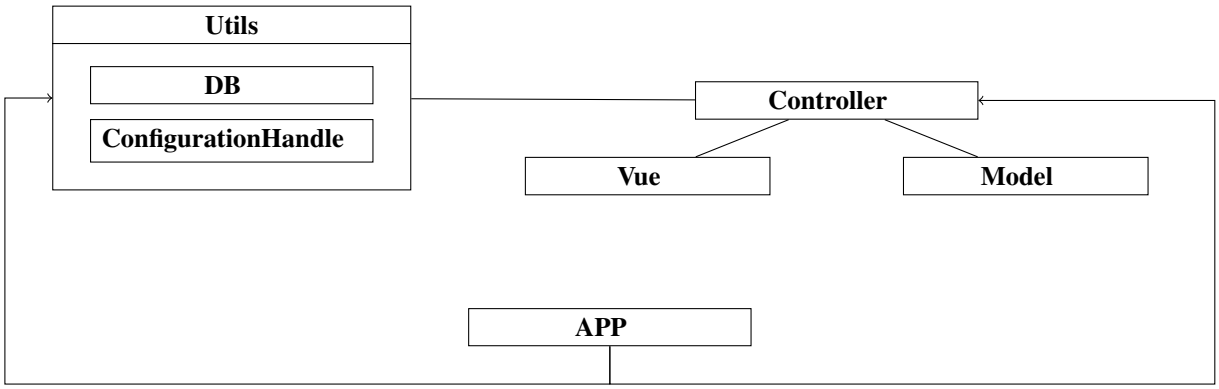
Les données liées à l'utilisateur sont plus volumineuses et nous avons donc utilisé une base de données. `SQLite` était la meilleure option pour nous. En effet, `SQLite` est une base de données relationnelle, simple et légère. Il n'y a pas besoin de lancer un serveur de base de données, mais nous profitons quand même de la puissance du SQL. Les quantités de données stockées étant raisonnables, cela correspondait parfaitement à nos besoins. De plus, des bibliothèques Java réalisent la communication très facilement.

5 Architecture de l'application

L'architecture de l'application est très importante afin de s'assurer d'une bonne évolution et maintenabilité. Ainsi, dans cette section nous présenterons l'architecture principale de l'application à travers un diagramme de paquetage, un diagramme de classe et un diagramme de séquence.
D'autres choix de conception serons présentés dans la section suivante *Choix de conception*

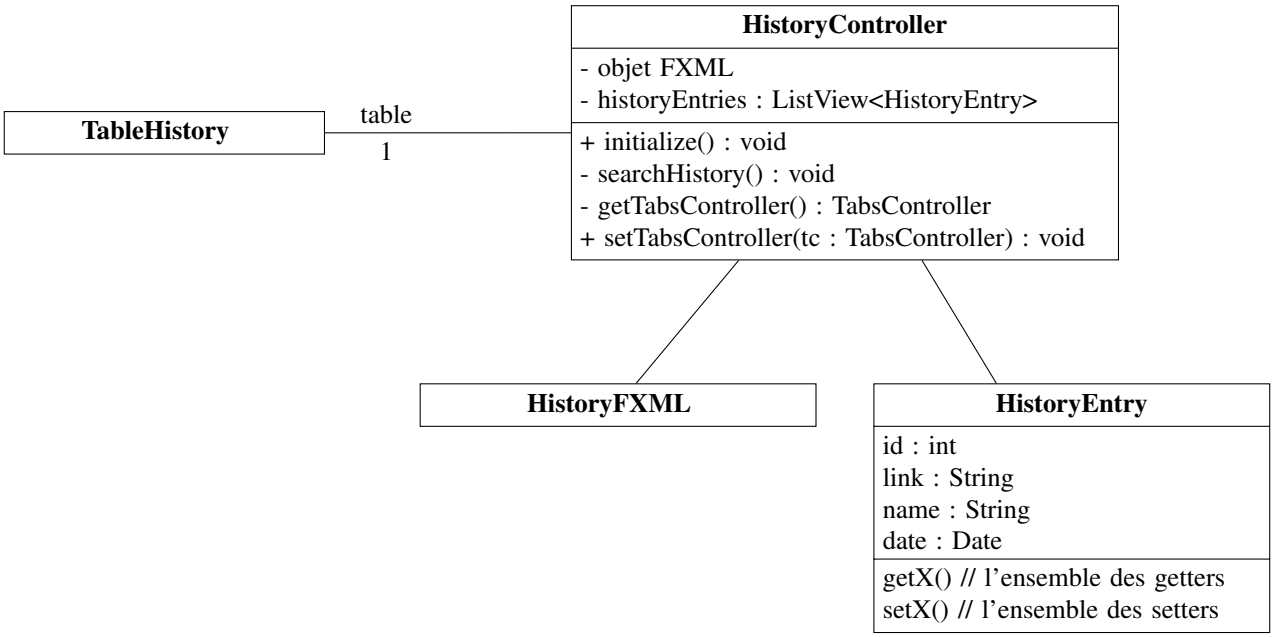
5.1 Organisation des paquetages

Les classes et interfaces de notre application ont été structurées dans nos différents paquetages en respectant le modèle MVC. Nous avons également un paquetage `Utils` qui englobe deux autres paquetages `DB` et `ConfigurationHandle` qui sont respectivement responsables de la gestion de la base de données et de la gestion des fichiers de configuration.



5.2 Exemple d'un diagramme de classe

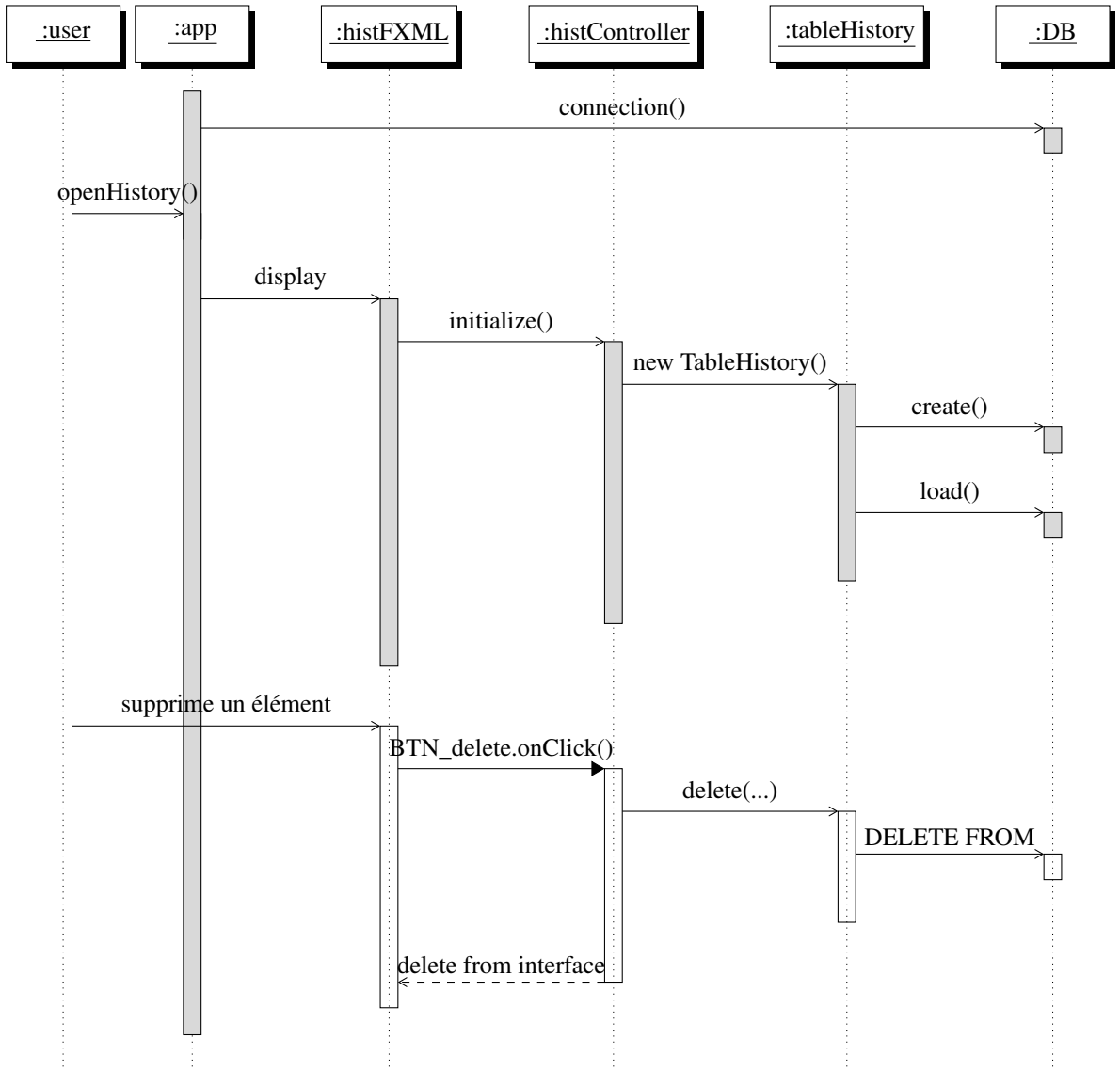
Les deux sections suivantes illustrent le service *Historique* à travers un diagramme de classe et un diagramme de séquence. Étant donné que, tous les services de l'application (favoris, onglets, cookies, paramètres) suivent la même architecture nous avons trouvé plus pertinent de ne représenter en détail qu'un seul service. Ainsi, ci-dessous vous retrouvez le diagramme de classe du service *Historique*.



L'implémentation des bases de données pour l'application est présentée plus en détail dans la section 6.1.3.

5.3 Exemple d'un diagramme de séquence

Afin de décrire l'enchainement des évènements entre `Model`, `Vue`, `Controller` et la base de données nous avons réalisé un diagramme de séquence. De même que la section précédente, ce diagramme de séquence est valable pour les favoris, les cookies, les onglets et les paramètres. Ici, nous l'illustrerons avec le service *Historique*.



6 Choix de conception

6.1 Stockage des données

Lors de la conception de l'application, nous nous sommes demandé s'il serait intéressant d'utiliser une base de données. En effet, nous avons besoin de stocker des informations persistantes telles que l'historique, les favoris, les cookies, etc. Différentes solutions s'offrent à nous pour réaliser le stockage. Nous pouvions soit utiliser un fichier texte soit utiliser une base de données.

6.1.1 Les fichiers texte

Ils ont l'avantage d'être simple à créer et à utiliser. De plus si nous utilisons une structure comme JSON ou XML il existe des librairies Java qui permettent de créer et récupérer les informations d'un fichier facilement.

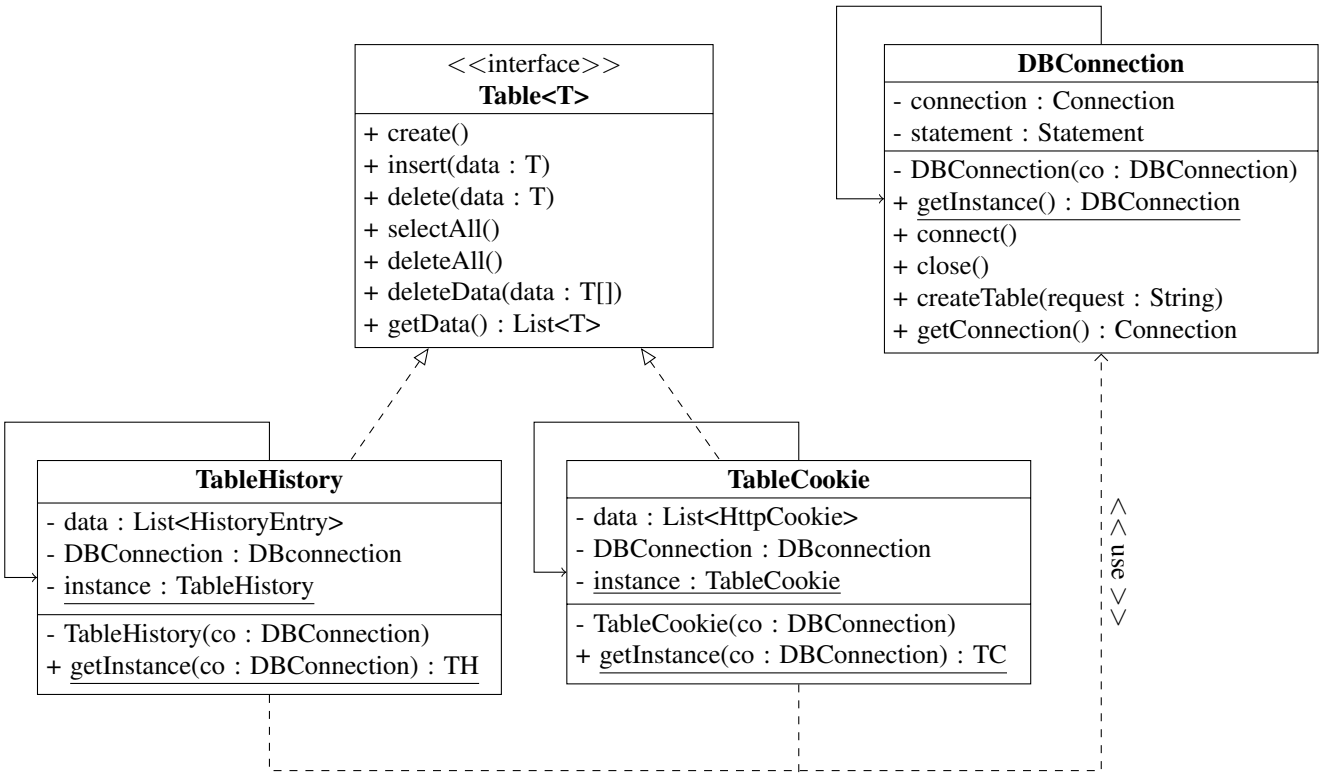
6.1.2 Une base de données

Est un outil conçu spécialement pour le stockage d'un grand nombre de données. Cependant, il est plus difficile de manipuler les bases de données en Java que les fichiers. De plus, nous devons également stocker les informations enregistrées en base de données sans que l'utilisateur n'ait besoin d'installer lui même un serveur de base de données.
Cette solution bien que plus compliquée à mettre en place a été retenue car elle présente de nombreux avantages une fois la technologie maîtrisée.

Ainsi, nous avons choisi une base de données `SQLite` qui permet d'avoir les informations stockées dans un fichier `.db` représentant la base de données. Ensuite, nous devions faire le lien entre nos classes Java (POJOs) et la table SQL. Une solution serait d'utiliser JavaEE et les annotations mais étant données que nous ne souhaitons pas charger le projet nous avons implémenté différents services qui permettent d'interagir avec la base de données.

6.1.3 Diagramme de classe

Ci-dessous nous représentons l'organisation des classes pour la gestions des tables de la base de données. La classe `DBConnection` permet de gérer l'accès à la base de données. Et, les implémentations de `Table` font le lien entre les objets Java (POJOs) et les tables de la base de données.



Un objet de type `DBConnection` est créé au lancement de l'application. Ensuite, pour pouvoir manipuler les tables, nous devons récupérer cet objet pour pouvoir effectuer des traitements dessus, c'est pour cela que nous le passons en paramètre de nos constructeurs `TableHistory` et `TableCookie`.

```
class HistoryController {

   private TableHistory table;

   @FXML
   intinialize() {
      table = TableHistory.getInstance(DBConnection.getInstance());
      ...
   }
}

class TableHistory implements Table<HistoryEntry> {

   private List<HistoryEntry> historyList;
   private DBConnection dbConnection;

   private static TableHistory instance;

   private TableHistory(DBConnection dbConnection) {
      this.dbConnection = dbConnection;
      this.historyList = new ArrayList<HistoryEntry>();

      this.create(); this.selectAll();
   }

   public static TableHistory getInstance(DBConnection dbConnection) {
      if(instance == null) {
         instance = new TableHistory(dbConnection);
      }
      return instance;
   }

   public void create() {
      /* utilisation de l'attribut dbConnection */
      dbConnection.createTable("CREATE TABLE ...");
   }
```

6.2 Stockage des configurations

Nous avons souhaité offrir à l'utilisateur la possibilité de configurer certaines options de l'application.

6.2.1 Configurer le navigateur

Premièrement, l'utilisateur peut enregistrer des configurations pour l'intégralité du navigateur web. Par exemple, le moteur de recherche par défaut, si l'utilisateur souhaite être en mode sombre, etc ...
De même que dans la section précédente, nous devons nous poser la question s'il faut utiliser une base de données ou un fichier de configuration. Pour ce cas, étant donné que les configurations possibles ne représentent pas un grand nombre de données à stocker, il est plus simple de manier un fichier formaté en JSON.

```
config.json
{
    engine: google // le moteur de recherche par défaut est Google
}
```

L'utilisateur a la possibilité de modifier la valeur du moteur de recherche par défaut depuis le menu *paramètre* de l'application. Ce qui aura pour conséquence de modifier la valeur dans le fichier `config.json`.

6.2.2 Configurer les moteurs recherches

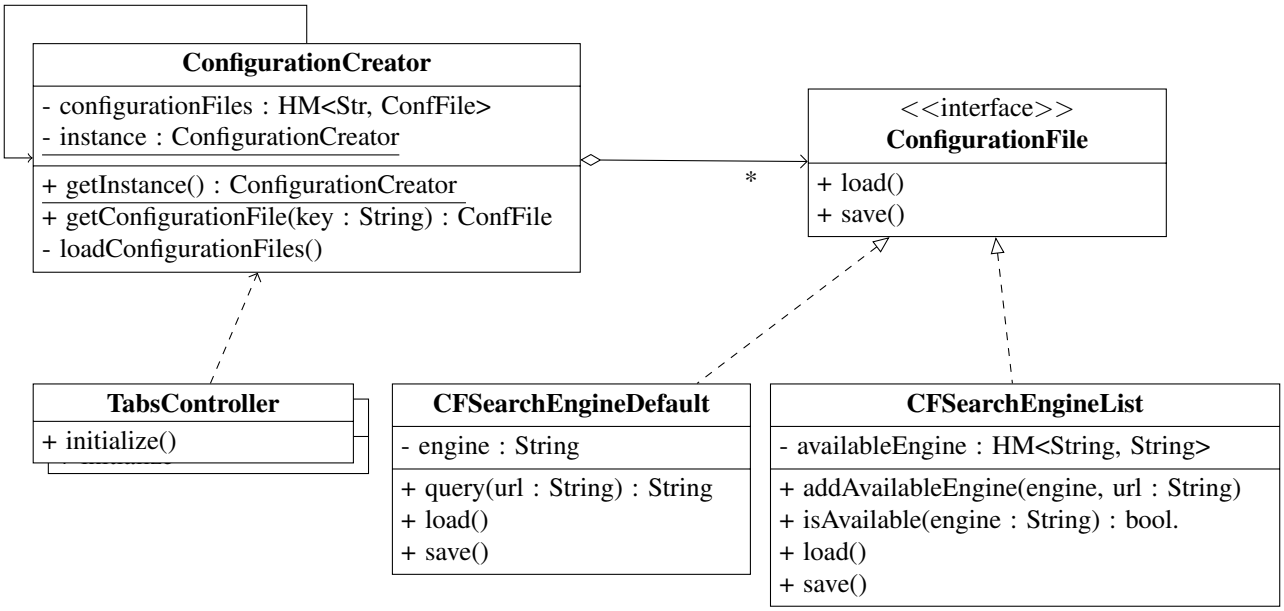
Ensuite, nous souhaitons que l'utilisateur puisse ajouter des moteurs de recherche ou des raccourcis de recherche au navigateur. Par exemple au lieu d'aller sur Youtube et d'effectuer une recherche, l'utilisateur peut directement écrire `@youtube nom_video` dans la barre de recherche.
Pour ce faire, de même que la configuration du navigateur nous utilisons un fichier JSON en associant un nom et une URL.

```
webengine.config
{
    youtube: https://youtube.com/result?search_query=
    google: https://google.com/search?q=
}
```

L'utilisateur a également la possibilité d'ajouter un raccourci `@` depuis les paramètres du navigateur.

6.2.3 Diagramme de classe

Nous présentons ci-dessous, un diagramme de classe sur la configuration de l'application



C'est dans la méthode `TabsController#initialize()` que nous instancions pour la première fois un objet de type `ConfigurationCreator`. Lors de la création de cet objet, nous exécutons la méthode`loadConfigurationFile()` qui va se charger de créer un nouvel objet pour chaque fichier de configuration avec une clé. Ainsi, par la suite nous pourrons récupérer cet objet grâce à la méthode `getConfigurationFile(key)`

```
class TabController {
   @FXML
   void initialize() {
      config = ConfigurationCreator.getInstance();
   }
}

class ConfigurationCreator {

   private ConfigurationCreator() {
      configurationsFiles = new HashMap<>();
      loadConfigurationFiles();
   }

   public static ConfigurationCreator getInstance() {
      /* Singleton */
   }

   public void loadConfigurationFiles() {
         configurationsFiles.put("configurationFileNavigator",
                                 new ConfigurationFileNavigator());
         configurationsFiles.put("configurationFileEngineSearch",
                                 new ConfigurationFileEngineSearch());
   }
}
```

Les autres contrôleurs pourront récupérer les objets associés aux fichiers de configuration.
C'est le cas dans `ParametersController` qui a besoin des fichiers de configuration du navigateur et des moteurs de recherche.

```
class ParametersController {
   @FXML
   private void initialize(){
      // get config from file
      ConfigurationCreator config = ConfigurationCreator.getInstance();

      CFSearchEngineDefault configNavigator =
            (CFSearchEngineDefault)
               config.getConfigurationFile("configurationFileNavigator");

      CFSearchEngineList configEngineSearch =
            (CFSearchEngineList)
               config.getConfigurationFile("configurationFileEngineSearch");

      /* manipulation des fichiers àtravers les deux objets */
}
```

6.3 Complémentarité des controlleurs

L'une des principales difficultés du projet était de pouvoir communiquer entre contrôleurs. En effet, nous avons défini un WebView qui permet d'avoir l'affichage des pages web. Mais lors de l'épique onglet, il a fallu faire en sorte que chaque onglet et une WebView différente. De plus, les boutons de la barre de navigation (suivant, précédent, etc) doivent communiquer seulement avec l'onglet et la WebView courante.

Pour résoudre ce problème, nous avons du imbriquer les différents controleurs afin de pouvoir manipuler plusieurs contrôleurs sur la même vue. Ainsi, la vue définie dans le `main.fxml` avec la fenêtre de navigation et la barre de navigation contient également une imbrication pour pouvoir utiliser la classe `TabsController`.
`<fx:include fx:id="tab" source="tabs.fxml" VBox.vgrow="ALWAYS" />`

6.3.1 Communication du fils vers le père

Et dans le contrôleur du `main.fxml` nous avons

```
@FXML // vaut toujours null si manquant
private TabsController tabController;

private void initialize() {
   tabController.setControlsController(this);
   ...
}
```

La méthode `TabsController.setControlsController` va permettre d'avoir connaissance de la barre de recherche dans la classe `TabsController`. Ainsi, lorsqu'on va changer d'onglet on va pouvoir également changer le nom dans la barre de recherche.

```
/**
 * Assign the window controller instance.
 * @param controlsController the window controller.
 */
public void setControlsController(WebViewController controlsController) {
   this.controlsController = controlsController;
   if (this.controlsController != null) {
      this.addressBar = this.controlsController.getAddressBar();
      this.progressBar = this.controlsController.getProgressBar();
   }
}
```

6.3.2 Communication du père vers le fils

Nous venons de donner l'instance du contrôleur parent au controleur fils. Maintenant, nous pouvons utiliser les méthodes définies dans `TabsController`. Par exemple en cliquant sur le bouton *ajouter un onglet* qui est défini dans `main.fxml` on va utiliser la méthode `TabsController#addNewTab()`.

```
newTabButton.setOnAction(new EventHandler<ActionEvent>() {
   @Override
   public void handle(ActionEvent event) {
      tabController.addNewTab(false);
      tabController.getCurrentTab().getTab().setText("Loading...");
      NavigationUtils.search(addressBar.getText(),
         tabController.getCurrentTab().getWebView().getEngine());
   }
});
```

7 Conclusion

7.1 Pistes d'amélioration

Avant de conclure ce projet, nous pouvons lister quelques pistes d'amélioration :

- Pour finaliser le projet, en se basant sur la vélocité, on estime qu'il nous faudrait deux semaines supplémentaires.
- De même, certains points d'effort ont été mal évalué. Mais, de meilleures estimations nécessitent une plus grande expérience.
- Une meilleure conception dès le début du projet aurait pu nous éviter d'avoir recours à du refactoring.
- Suivre jusqu'à la fin du projet le workflow détaillé sur le GitHub. Car à la fin du projet nous ne nous relisions moins qu'au début (prise de confiance).
- Enfin, il manque des tests unitaires car l'architecture est très simple et les tests se baseraient principalement sur l'interface graphique.

7.2 Conclusion sur l'agilité

Le fait d'avoir d'utilisé l'agilité pour ce projet nous a prouvé que ce concept fonctionne bien. En effet, au fur et à mesure des itérations on arrivait à prioriser les tâches et à apporter des modifications sur des points d'effort et des valeurs métier. Ensuite, la répartition des tâches était équitable. Chacun pouvait travailler sur son service indépendamment puis ensuite le mettre en commun via Git.
Enfin, nous finissions chaque itération par une rétrospective pour se remettre en question et continuer à s'améliorer.

7.3 Conclusion du projet

Notre backlog était très ambitieux, et comportait des épiques et fonctionnalités très poussées. Nous ne sommes pas parvenus à la fin de celui-ci. Les fonctionnalités essentielles, avec les plus grandes valeurs métiers (de 6 à 10) ont été choisies et terminées. Heureusement pour nous, les fonctionnalités les plus difficiles, avec les points d'efforts les plus hauts étaient souvent des fonctionnalités accessoires, ou en tout cas moins critiques (des valeurs métiers de 1 à 5), comme par exemple les épiques concernant le gestionnaire de mot de passes (4 user stories). Nous sommes confiants sur le fait qu'avec un ou deux sprints supplémentaires l'ensemble du backlog aurait pu être réalisé.

Nous avons beaucoup apprécié l'opportunité d'enfin mettre en pratique l'agilité pour ce projet au travers de la méthode SCRUM. Nous avons pu faire l'expérience de sa valeur ajoutée : par exemple le fait d'avoir une application fonctionnelle à la fin de chaque itération était très gratifiant et nous encourageait à continuer. Fonctionner par itérations limitait la portée de nos décisions et nous permettait de corriger les erreurs (par exemple mauvaises estimations de user stories). Le fait de beaucoup communiquer lors des retrospective, de faire rapport de son progrès et surtout de ses difficultés a grandement contribué à instaurer un climat d'entraide plutôt qu'une atmosphère de compétition à qui finirait son ticket le premier.

Enfin, pour finir de nous imprégner jusqu'au bout de la philosophie agile, nous étions bien-sûr obligés de faire des activités extra-curriculaires de team-building. Nous avons donc organisé avec une autre équipe un week-end de cohésion pour souder nos liens afin de nous aider à mieux avancer ensemble sur le projet.