

**TP3
RÉALISATION DU PROJET ET
APPLICATIONS DE PATRONS GoF**

**TRAVAIL PRÉSENTÉ À :
JACQUES BERGER**

**GÉNIE LOGICIEL: CONCEPTION
INF-5153
GROUPE-020**

PAR

**OLIVIER CAMPEAU - CAMO23028700
PHILIPPE CHARTRAND - CHAP24059408
THOMAS CORBEIL - CORT16109704
HUGO TWIGG-CÔTÉ - TWIH25048700**

**UNIVERSITÉ DU QUÉBEC À MONTRÉAL
25 JUILLET 2019**

TABLES DES MATIÈRES

| | |
|---|-----------|
| Introduction | 1 |
| 1. Modifications apportées | 1 |
| 1.1 Système central | 1 |
| 1.2 AppMedecin | 2 |
| 2. Patrons Gang of Four | 3 |
| 2.1 Singleton | 3 |
| 2.2 Facade | 5 |
| 2.3 Factory Method | 7 |
| 2.4 Prototype | 9 |
| 3. Instruction pour utiliser l'application | 10 |

Introduction

Ce document est la troisième remise du projet de session sur la conception d'une application pour gérer des dossiers médicaux. D'abord, nous allons aborder les modifications apportées à la conception durant la phase de réalisation. Ensuite, nous présenterons quatre (4) patrons *Gang of Four* utilisés durant la réalisation. Nous terminerons avec les instructions expliquant comment démarrer l'application pour laquelle le code également été remis.

1. Modifications apportées

Les modifications apportées à la conception présentée au TP2 sont décrites ci-dessous. Ces modifications ne comprennent pas les patrons de conception Gang of Four ajoutés à la conception. Ces modifications sont décrites à la section 2 (Patrons Gang of Four).

1.1 Système central

La classe `ServiceAuthentification` a été modifiée afin de sortir les deux méthodes privées suivantes de la classe: `genererToken()` et `creerSession()`. Ces méthodes étaient redondantes puisque nous pouvons accomplir la même tâche en utilisant la classe `ServiceGestionFichier`. Cela permettait du même coup à la classe `ServiceAuthentification` d'avoir une meilleure cohésion.

Un problème que nous avons rencontré est que n'avions pas d'objet dans lequel retourner le token une fois l'utilisateur authentifié par la classe `ServiceAuthentification`. Nous avons donc modifié la classe `ReponseAuthentification` afin d'y ajouter un attribut token.

Un autre problème que nous avons rencontré est que nous n'avions pas moyen avec seulement le token, dans la classe `FiltrePermission`, de connaître le nom de l'utilisateur sans violer le principe d'architecture en couche. Et nous avons besoin de l'identifiant pour récupérer ses permissions. Autrement dit, dans notre architecture, la seule façon d'obtenir l'identifiant de l'utilisateur dans cette classe était de communiquer avec une couche plus profonde que la suivante. Nous avons réglé ce problème en ajoutant la méthode `obtenirSession()` dans la classe `ServiceGestionSession`. Avec cette méthode, nous pouvions obtenir un objet `Session` qui contient l'identifiant de l'utilisateur.

Enfin, un autre problème que nous avons rencontré est que les objets (les tables) dans la base de données ne correspondaient pas tout le temps exactement aux objets du domaine. Par exemple, dans la base de données les visites médicales ont une référence sur le dossier médical auquel elles appartiennent, mais cette référence n'est pas nécessaire dans l'objet du domaine. Nous avons donc besoin d'objets intermédiaires entre les objets de la base de données et les objets du domaine afin de pouvoir les faire correspondre. Nous avons donc créé des objets DTO (*Data Transfer Object*) et des classes utilitaires *Mapper* afin de pouvoir établir une correspondance entre les objets de la base de données et les objets du domaine.

1.2 AppMedecin

Dans le TP2, nous avons oublié d'ajouter la méthode `annulerDerniereModification()` dans la classe `GestionnaireDossierActif`. Nous l'avons donc ajouté.

Nous avons aussi décidé de ne pas créer de nouveaux objets du domaine spécifiquement pour la vue, mais de plutôt travailler directement avec les objets du domaine. Cela était plus simple au niveau de la conception. Autrement, nous aurions eu deux types d'objets trop semblables et il y aurait eu de la duplication de code. Ainsi, la classe `DossierVueSpecialisteSante` et toutes les classes étant référées par la classe `DossierPresentationSpecialisteSante` ont été écartée de la réalisation.

Nous avons réduit le temps de «batching» pour les modifications sauvegardées automatiquement de 5 secondes à 1 secondes, car nous trouvions que avec 5 secondes de délais, trop de modifications étaient annulées en même temps lors d'une annulation. Nous croyons que 1 seconde de délais au lieu de 5 augmente l'expérience utilisateur.

Nous avons aussi simplifié les classes de la couche de présentation. Nous avons regroupé les classes `FenetreListeVisiteMedicaleSpecialisteSante`, `FenetreCreerVisiteSpecialisteSante` et `FenetreVisiteMedicaleSpecialisteSante` une seule et même classe. Cela simplifiait grandement le partage d'information sur les visites à afficher. Puisque ces fenêtres utilisent toutes les mêmes données, nous avons utilisé le patron «Expert en information» et nous avons mis la fenêtre de l'affichage d'une visite dans la même classe que la fenêtre de la liste des visites, car c'est ce composant qui contient l'information à afficher. Nous avons fait de même avec la fenêtre `FenetreCreerVisiteSpecialisteSante` qui ne devait contenir qu'un bouton permettant de créer une nouvelle visite.

La même logique a été appliquée aux classes

`FenetreListeAntecedentSpecialisteSante`,
`FenetreAntecedentSpecialisteSante` et

`FenetreCreerAntecedentSpecialisteSante`, qui font exactement la même chose que les classes citées précédemment, à l'exception près qu'elles gèrent les antécédents médicaux au lieu des visites médicales.

2. Patrons Gang of Four

Cette section décrit les quatre (4) patrons *Gang of Four* que nous avons utilisé dans notre conception lors de la réalisation du projet. Pour chaque patron, nous expliquons pourquoi nous avons utilisé ce patron, et nous fournissons un diagramme de classe et un diagramme de séquence montrant l'implémentation l'utilisation du patron.

2.1 Singleton

Pour éviter de créer plusieurs connexions à la base de données et de perdre le contrôle, nous avons décidé que l'objet qui renferme la connexion à la base de données serait un Singleton. Dans notre application, il y a deux types d'objets de connexion: un objet avec une seule connexion, et un objet avec un pool de connexion. La classe `SQLiteConnectionCreator` peut instancier un des deux. Le diagramme de classe suivant montre qu'en fait, les deux objets `SQLiteSingleConnection` et `SQLitePooledConnection` sont des Singleton, et que la classe `SQLiteConnectionCreator` peut instancier les deux (idéalement un ou l'autre). Le diagramme de séquence qui suit montre que la classe `SQLiteConnectionCreator` à un choix à faire entre les deux.

Diagramme de classe du patron Singleton:

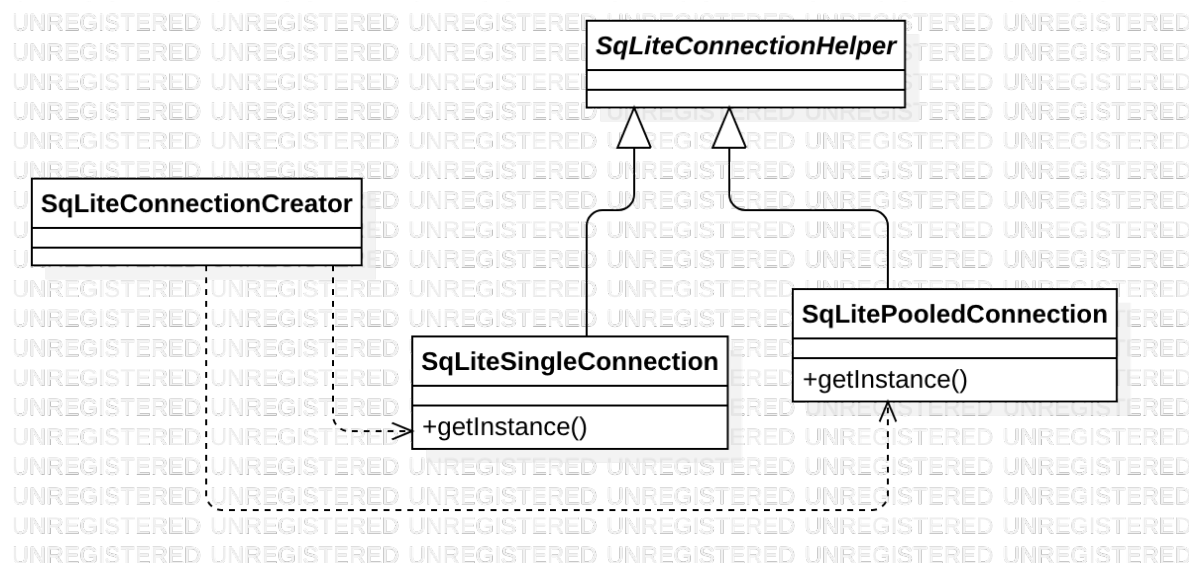
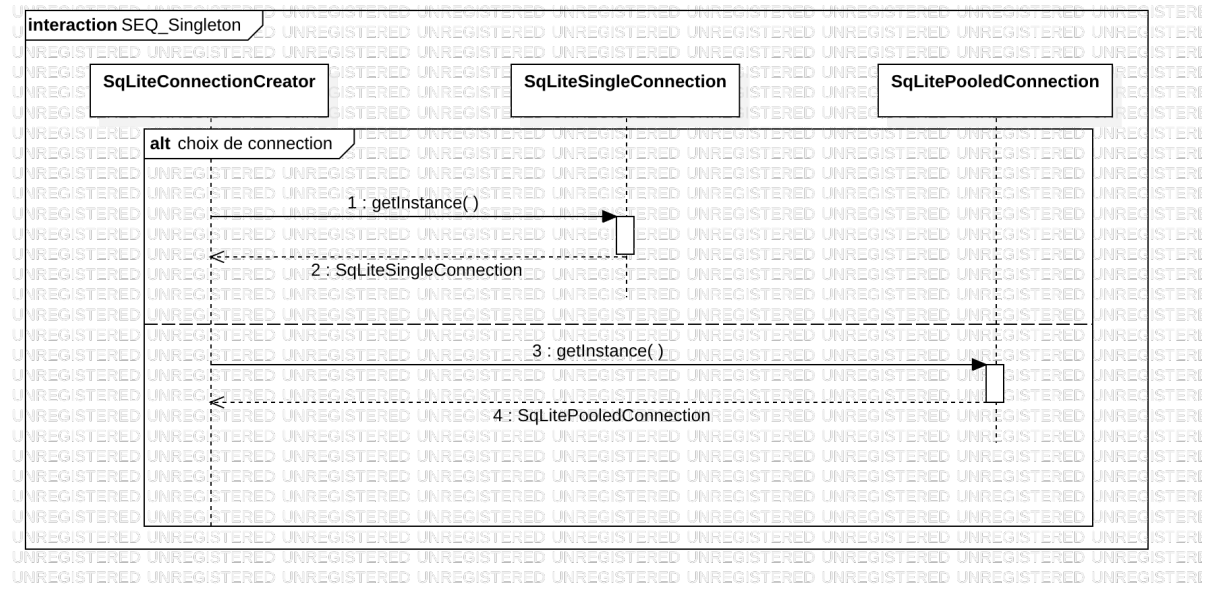


Diagramme de séquence du patron Singleton:



2.2 Facade

Tel qu'expliqué ci-haut, pour faire la correspondance entre les objets de la base de données et les objets du domaine, nous avons créé des classes utilitaires Mapper. Or, pour faire correspondre un dossier médical de la base de données avec un objet du domaine de type `DossierMedical`, plusieurs conversions d'objets étaient nécessaires. En effet, un dossier médical contient un patient, des antécédents médicaux et des visites médicales, et ces dernières contiennent à leur tour d'autres objets comme des coordonnées. Or, tous ces objets sont des tables différentes dans la base de données. Ainsi, nous avons créé une façade, intitulée `DossierMedicalMapper`, qui offre une interface très simple avec seulement deux fonctions: `dtoToDossier()` qui se charge de convertir un objet `DTODossierMedical` vers l'objet du domaine `DossierMedical`, et la fonction `mapToDto()` qui fait l'inverse, soit convertir un objet `DossierMedical` vers un `DTODossierMedical`. La façade `DossierMedicalMapper` se charge de faire appel aux autres classes utilitaires *Mapper* afin d'effectuer les multiples opérations nécessaires pour effectuer la conversion de dossier. Le diagramme de séquence qui suit démontre l'utilisation de la façade pour convertir un objet `DTODossierMedical` vers un objet de type `DossierMedical`.

Diagramme de classe du patron Facade:

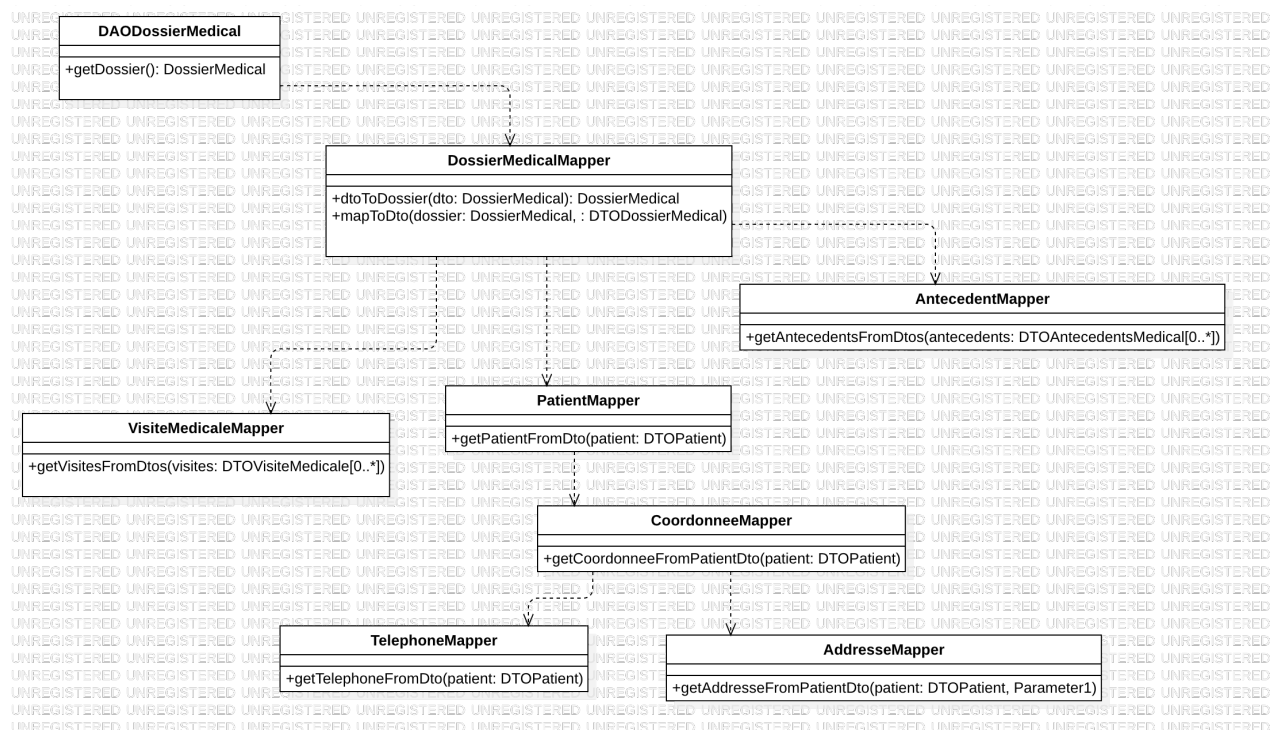
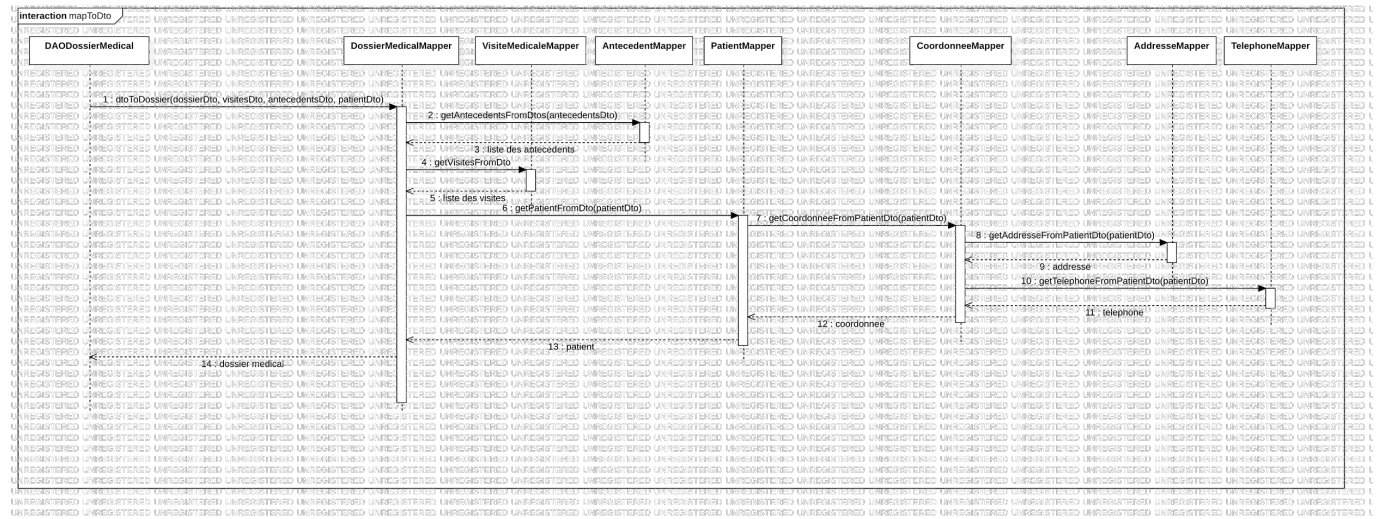


Diagramme de séquence du patron Facade:



2.3 Factory Method

Afin de pouvoir plus facilement créer un objet de type `JDBCConnectionHelper` qui permet de se connecter à la base de données, nous avons utilisé le patron de conception *Factory Method*. Il y a donc une interface générique `JDBCConnectionCreator` qui permet de créer un objet de type `JDBCConnectionHelper`. Nous avons donc implémenté l'interface `JDBCConnectionCreator` avec un objet de type `SQLiteConnectionCreator` qui permet de créer des objets de connexion pour une base de données SQLite, soit un objet `SQLiteSingleConnection` ou un objet `SQLitePooledConnection`. Il suffit de passer un type énumératif `ConnectionType` à la fonction `createConnection()` de la classe qui implémente l'interface `JDBCConnectionCreator` (dans notre cas c'est `SQLiteConnectionCreator`) afin d'obtenir l'objet de connexion désiré.

Diagramme de classe du patron Factory Method:

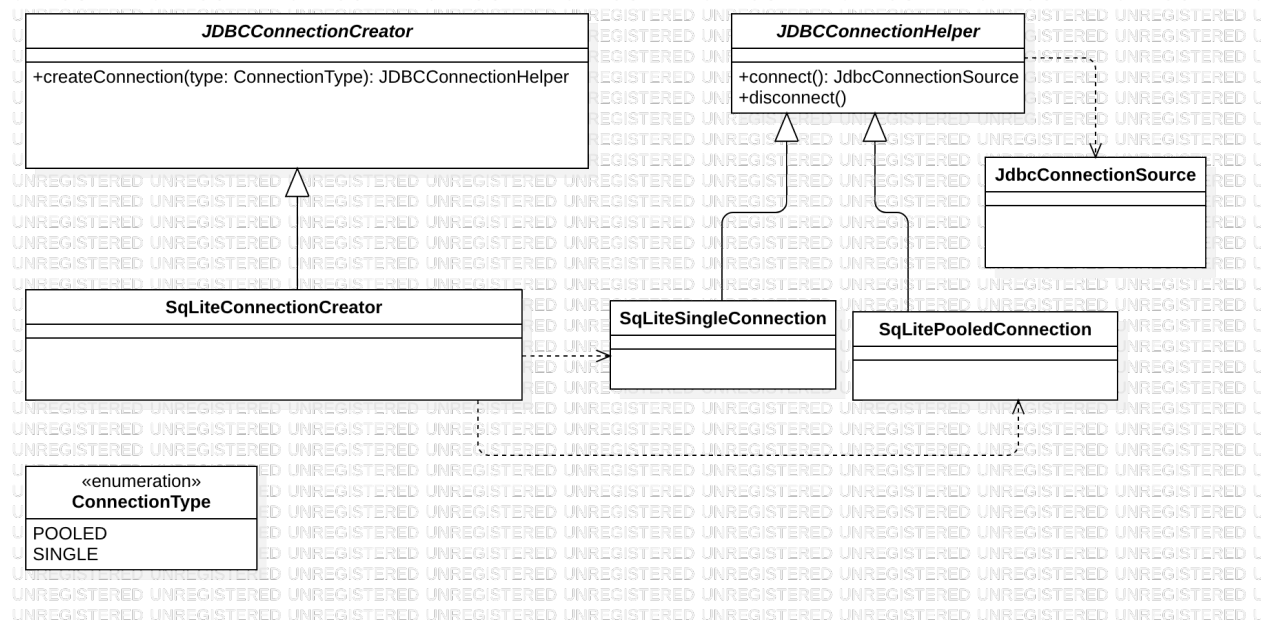
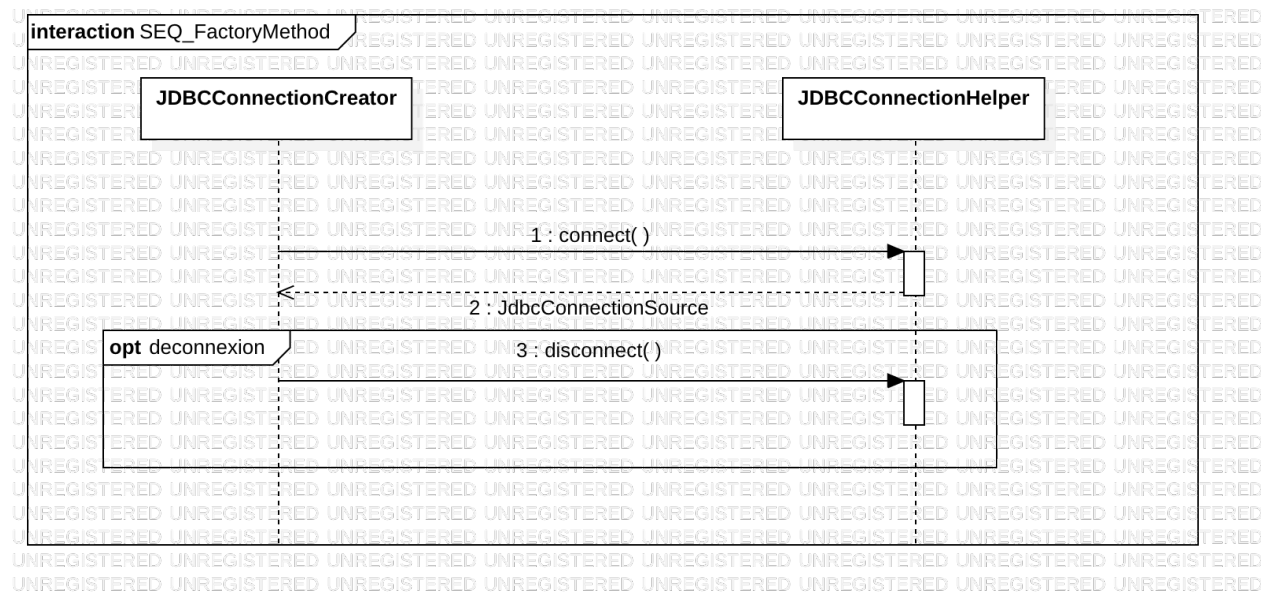


Diagramme de séquence du patron Factory Method:



2.4 Prototype

Un des problèmes que nous avons rencontrés dans la réalisation est que lorsqu'une visite médicale est créée, elle doit toujours être liée à l'établissement de santé dans laquelle elle a été créée. En effet, un médecin ne peut pas créer une visite médicale dans un établissement de santé en particulier, mais indiqué que cette visite a eu lieu ailleurs. Pour nous assurer de cela, nous avons utilisé le patron Prototype. La classe *GestionnaireDossierActif*, lorsqu'elle est initialisée, va chercher le nom de l'établissement de santé dans la configuration de l'application et créer un objet *VisitePrototype* avec toutes les données de l'établissement de santé déjà enregistrée. Ainsi, pour créer une nouvelle visite, il suffit de cloner le prototype d'une visite médicale, et cette visite contiendra les informations relatives à l'établissement de santé où se trouve le médecin.

Diagramme de classe du patron Prototype:

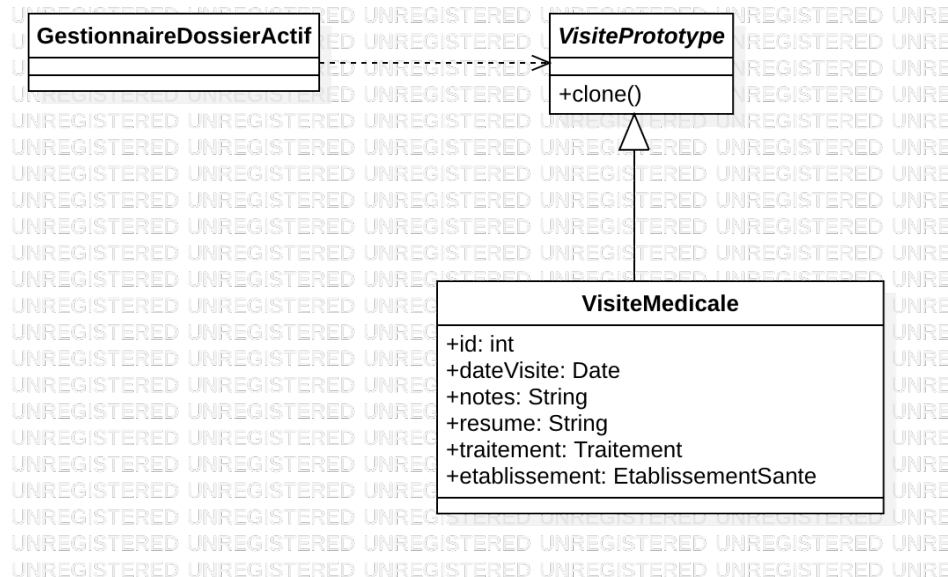
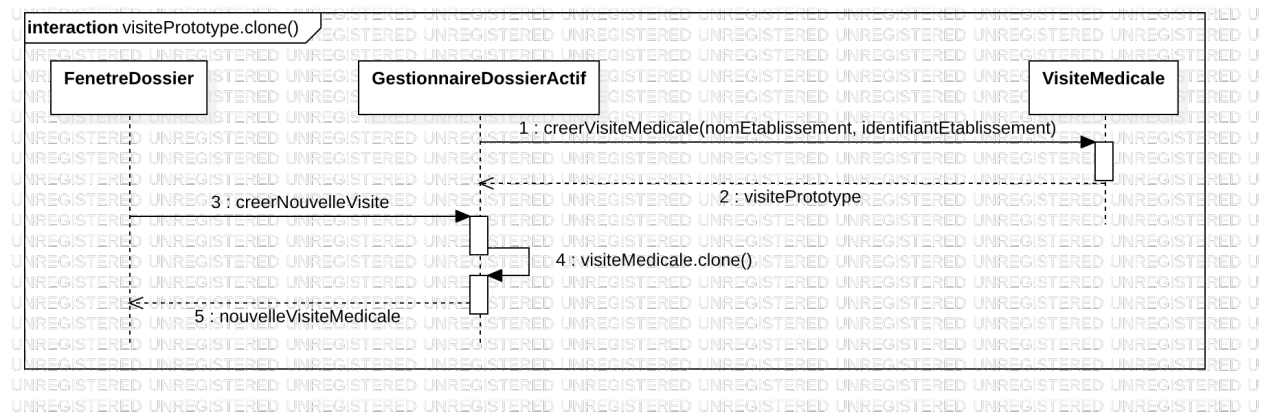


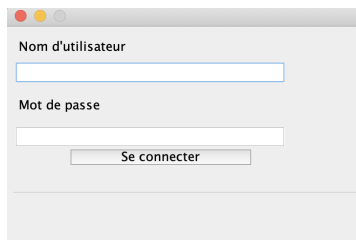
Diagramme de séquence du patron Prototype:



3. Instruction pour utiliser l'application

Pour savoir comment démarrer l'application, référez-vous au fichier README.md inclus avec le code source. Pour se connecter à l'application, utilisez le formulaire de connexion suivant et entrez les informations suivantes et cliquez sur le bouton « Se connecter » :

- username: medecin1
- mot de passe: admin.



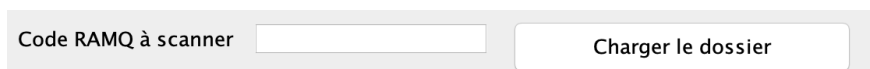
Form de connexion :

Nom d'utilisateur

Mot de passe

Se connecter

Ensuite, il faut entrer le numéro d'un dossier dans la barre de recherche suivante et cliquer sur le bouton « Charger le dossier ». Les numéros disponibles sont dans le fichier README.md.



Code RAMQ à scanner

Charger le dossier

Ensuite, lorsque le dossier est chargé, il suffit de le consulter et de le modifier avec l'interface suivante. Pour afficher les données d'une visite médicale, il suffit de cliquer sur une visite dans la liste des visites. Chaque champ de texte de la visite est modifiable et les modifications sont enregistrées automatiquement. Idem pour les antécédents médicaux.



Visites médicales

Ajouter une visite

2019-07-05 22:52:49 : Clinique Super Bonne Inc.
2019-07-05 22:52:49 : Clinique Super Bonne Inc.
2019-07-05 22:52:49 : Clinique Super Bonne Inc.
2019-07-17 23:30:06 : Clinique Super Malade Inc.
2019-07-17 23:48:41 : Clinique Super Malade Inc.
2019-07-18 08:49:30 : Clinique Super Malade Inc.

Notes

Rien a signalé

Resume

Blablabla

Diagnostic

Traitement

Medicament

Antecedent médicaux

Ajouter un antécédent

2019-07-05 22:52:09 : Grippe d'homme
2019-07-08 14:59:24 : Grippe d'homme
2019-07-08 14:59:24 : ceci est un update antecedent
2019-07-17 19:28:26 :
2019-07-17 19:31:34 : Grippe d'homme
2019-07-18 08:58:22 : d
2019-07-18 09:04:21 : dedfhabs
2019-07-18 09:07:38 :

Début de la maladie

Fin de la maladie

Diagnostic

Traitement

Medicament

Annuler les dernières modifications