Plan :

Introduction / Présentation

1. Analyse
2. Conception préliminaire
3. Conception détaillée
4. Tests unitaires

Conclusion personnelle et globale

**Introduction / Présentation**

Le projet concerne les zones de stockage et des systèmes de consignes automatiques. Les consignes automatiques sont des grandes armoires compartimentées en casiers, disposées dans les gares ou les aéroports. C'est la société Logibag SAS qui est chargée de concevoir et de réaliser les consignes automatiques.

Il existe plusieurs catégories de consignes automatiques. Il y a notamment les consignes "business to business", utilisées par les techniciens qui peuvent faire déposer par un livreur du matériel partageable. Les consignes postales, qui, situées au cœur des villes, sont utilisés pour les clients afin de récupérer des colis entreposés par des livreurs c’est dans la cadre des consignes postales que ce projet intervient.

Ces colis sont récupérés par les clients, après qu'ils aient reçu, par exemple par SMS ou mail un code de retrait. Enfin, il y a les consignes à bagages. Ces consignes sont trouvables dans les gares ou les aéroports. Ce sont de systèmes classiques en libre service. Le paiement s'effectue en fonction de la durée.

L’ensemble du système est constitué :

* de l'ordinateur console ;
* de consignes de différentes tailles d’un nombre maximum de 255 ;
* de cartes DCB (reliées au module technique par bus CAN) qui contrôlent les portes des consignes ;
* d’un serveur LMS (Locker Management Server) qui centralise les bases de données de tous les postes.

Le système Linux de l’ordinateur console contrôle via un logiciel l'ouverture et la fermeture des consignes, gère l'interface utilisateur qui peut varier sous quatre catégories: l’installateur, l'opérateur de maintenance, le livreur et le client. L’installateur s’occupe uniquement de la configuration du système, l’opérateur de la configuration et de la maintenance. Le livreur peut déposer ou retirer des colis. Ce dernier cas peut se présenter si le client n’a pas retiré à temps le colis selon un délai.

Le projet a pour but de remplacer le système vieillissant qui n’est plus adapté aux technologies actuelles. Il s’agit donc à la fois d’installer un système d’exploitation plus récent et de développer une nouvelle version de l’application appelée Locker Control (LC). Ce logiciel sera le cœur de gestion du système. L’ajout de L’assistance en ligne et un serveur de mise à jour.

1. **Analyse**

* Evolution de l’application en C++ en utilisant QT4 ;

Qt est un ensemble de bibliothèques de développement multiplateforme et multi-langages dont le C++. C’est un outil qui permet de développer des programmes très efficacement, il offre une simplicité dans la programmation et dans la création des interfaces graphiques.

* Ajout d’une interface de configuration et de maintenance ;
* Développement d’une fonction d’assistance en ligne ;
* Communication avec la base de données locale de type SQLite qui sera décrit plus tard ;
* Gestion des comptes utilisateurs par la communication avec la base de données, le serveur LMS pour le code de retrait envoyé aux clients en attente de récupérer un ou des colis.
* Nous avons prévu d’utiliser Git, un logiciel de gestion de versions décentralisé. Cet outil permettra à chaque membre du groupe d’apporter sa contribution en matière de codage de l’application. En dépôt distant nous utiliserons GitHub, ce qui permet d’avoir accès au code depuis n’importe quel poste.
* Pour réaliser les tests unitaires nous utiliserons la librairie QTestLib. Elle permet de créer un programme qui réalise les tests et peut être automatique avec un rapport d’erreurs par exemple.

Voici le diagramme de cas d’utilisation  qui regroupe l’ensemble des acteurs et leur rôle dans le système à mettre en œuvre:

* Client

A l’aide d’un code de retrait, fourni par mail ou SMS au préalable par le serveur LMS, le client peut retirer ses colis. En cas de problème lors de du retrait le client peut demander assistance.

* Livreur

Après avoir été authentifié, le livreur peut déposer dans les consignes des colis pour différents clients. Si au moment de déposer le colis, la consigne est défectueuse, il a deux choix : soit il change de consigne, soit il demande assistance auprès de l’assistance en ligne. Le livreur est en droit de retirer les colis qui ont été oubliés par les clients pour libérer des consignes.

* Opérateur

Après avoir été identifié, l’opérateur pourra modifier les paramètres système. Il réalise le test des portes des consignes : supprimer, ajouter, activer ou désactiver des portes et enfin mettre à jour la base de données.

* Installateur

Il est chargé de l’installation et de la configuration du site et notamment les paramètres du système et des consignes.

* Gestionnaire

Cette partie est hors projet, cela consiste à mettre à jour les données du serveur LMS, qui contient l’état des consignes, la liste des livreurs et des clients et les codes de retrait des colis.

* Assistant

L’assistant offre ses services en cas de besoin au client et/ou au livreur et peut de manière optionnelle intervenir dans la livraison ou dans le retrait pour résoudre un problème d’ouverture ou de fermeture des consignes.

**II . Conception préliminaire**

Plan :

* Basé sur la librairie SQLite
* Conception de la base de données
  + Règles de gestions
  + Modélisation du MCD
  + Création du MLD
  + Générer le SQL à partir du MLD

La conception d’une base de données se base sur quatre étapes fondamentales :

* L’élaboration de règles de gestions. Il faut recueillir les besoins des futurs utilisateurs de l’application. Et à partir de ces besoins, être en mesure d'établir les règles de gestion des données à conserver.
* Le Modèle Conceptuel de Données (MCD). Il s'agit d’une représentation graphique et structurée des informations mémorisées par la future base de données. Le MCD est basé sur deux notions principales : les entités et les associations.
* Le modèle logique de données (MLD) est composé uniquement de ce que l'on appelle des relations. Ces relations sont à la fois issues des entités du MCD mais aussi d'associations, dans certains cas. Ces relations nous permettront par la suite de créer nos tables au niveau physique (SQL).
* Générer la construction de la base de données avec le SQL grâce au MLD.

**Régles de gestion :**

Chaque entité est unique et est décrite par un ensemble de propriétés encore appelées attributs ou caractéristiques. Une des propriétés de l'entité est l'identifiant. Cette propriété doit posséder des occurrences uniques et doit être source des dépendances fonctionnelles avec toutes les autres propriétés de l'entité. Bien souvent, on utilise une donnée de type entier qui s'incrémente pour chaque occurrence, ou encore un code unique spécifique du contexte.

À partir de cette entité, on peut retrouver la règle de gestion suivante :

Un utilisateur est identifié par un mot de passe, constitué d’un identifiant de société, d’un identifiant personnel enfin d’un code secret et est caractérisé par un rôle sur le système.

D’après le cahier des charges on peut recenser 7 autres entités en plus de l’entité « User » que l’on peut représenter avec les règles de gestions suivantes.

* Une console de gestion est identifiée par un code unique et est caractérisé par un nombre de consignes contrôlées.
* Une carte de contrôle DCB (Door Control Board) est identifiée par un numéro unique et est caractérisée par le contrôle de deux verrous.
* Un colis est identifié par un code unique et est caractérisé par celui-ci.
* Une consigne est identifiée par un numéro unique et est caractérisé par sa taille et 4 états possibles Libre/Occupée et Endommagée/Fonctionnelle.
* Un verrou est identifié par un numéro unique (identifiant bus CAN) et est caractérisé par 4 états possibles Ouvert/Fermé et Verrouillé/Déverrouillé.
* Enfin une sauvegarde des actions effectuées se caractérise par un message (courte description de l’action qui sera effectué sur la base de données), le nom de la table modifié et un horodateur de l’action.
* Une carte de contrôle DCB peut contrôler un ou deux verrous, chaque verrou est contrôler par une et une seule carte DCB.
* Une consigne peut être verrouillée par un ou deux verrous, chaque verrou ne contrôle qu’une seule consigne.
* Une consigne peut contenir aucun, un ou plusieurs colis, un colis ne peut appartenir qu’à une et une seule consigne
* Un colis peut être lié à un et un seul code de retrait, un code de retrait peut être lié à un ou plusieurs colis.

**MCD**

Une association définit un lien entre une ou plusieurs entités. En effet, la définition de liens entre entités permet de traduire une partie des règles de gestion qui n'ont pas été satisfaites par la simple définition des entités.

Généralement le nom de l'association est un verbe définissant le lien entre les entités qui sont reliées par cette dernière.

Avec toutes ces connaissances, il nous est donc possible d'élaborer le MCD complet à partir des données présentes dans les règles de gestion énoncée précédemment :

**MCD -> MLD**

Le modèle logique de données (MLD) est composé uniquement de ce que l'on appelle des relations. Ces relations sont à la fois issues des entités du MCD mais aussi d'associations, dans certains cas. Ces relations nous permettront par la suite de créer nos tables au niveau physique (SQL).

Une relation est composée d'attributs. Ces attributs sont des données élémentaires issues des propriétés des différentes entités mais aussi des identifiants et des données portées par certaines associations.

Une relation possède un nom qui correspond en général à celui de l'entité ou de l'association qui lui correspond. Elle possède aussi une clef primaire qui permet d'identifier sans ambiguïté chaque occurrence de cette relation. La clef primaire peut être composée d'un ou plusieurs attributs, il s'agit d'une implantation de la notion d'identifiant des entités et associations qui se répercute au niveau relationnel.

En suivant des règles de passage du MCD au MLD, on peut en déduire facilement les changements éventuels à effectuer sur notre base de données.

Règle 1 - conversion d'une entité :

* En règle générale, toute entité du MCD devient une relation dont la clef est l'identifiant de cette entité. Chaque propriété de l'entité devient un attribut de la relation correspondante.

Règle 2 - conversion d'associations n'ayant que des cardinalités de type 0/1, N

* Une association ayant des cardinalités « 0, N » ou « 1, N » de part et d'autre devient une relation dont la clef est constituée des identifiants des entités reliées par cette association. Ces identifiants seront donc également des clefs étrangères respectives. On parle de **relations associatives**.

Règle 3 - conversion des associations ayant au moins une cardinalité de type 1,1

* Plusieurs possibilités s'offrent à nous pour ce cas de figure. La règle de conversion la plus répandue aujourd'hui est d'ajouter une clef étrangère dans la relation qui correspond à l'entité se situant du côté de cette cardinalité 1,1. Cette clef étrangère fera donc référence à la clef de la relation correspondant à la seconde entité reliée par l'association.

Règle 4 - conversion des associations ayant au moins une cardinalité de type 0,1 (et dont les autres cardinalités sont de type 0,1/N)

* De même que pour les cardinalités 1,1, une association ayant une cardinalité 0,1 doit être binaire, et les deux mêmes possibilités s'offrent à nous :
  + Créer la clef étrangère dans la relation correspondant à l'entité du côté de la cardinalité 0,1. Rappelons que dans ce cas, l'association ne peut pas être porteuse de données.
  + Créer une relation associative qui serait identifié de la même façon que pour une cardinalité 1,1.

Cependant, dans le cadre d'une cardinalité 0,1, nous verrons qu'il n'est pas toujours préférable de privilégier la première méthode comme c'est le cas pour une cardinalité 1,1.

Avec toutes ces connaissances, il nous est donc possible d'élaborer le MLD complet à partir du MCD et des règles de passages du MCD au MLD.

On peut apercevoir les nouveaux champs à créer issue des associations entre les tables. La dernière étape est maintenant de passer du MLD au format SQL.

**MLD -> SQL**

Certain outils comme MySQL Workbench proposent de passer au format SQL directement par le MLD ci-dessus. Une fois au format SQL, la base de données est prête à être construite par la librairie SQLite en exécutant les requêtes SQL. Ce fichier contenant les requêtes SQL peut aussi servir à une remise en état initial de la base de données.

**Inplémentation de la classe CSQLite\_Local\_DB**

Une seule instance de l’objet CSQLite\_Local\_DB doit être créé, les classes qui souhaitent avoir un accès sur la base de données se verront attribué un pointeur ou équivalent sur celui-ci. Ce mécanisme évite d’avoir différents objets qui interagissent avec la même base de données. Le diagramme de classe suivant explique l’implémentation nécessaire à effectuer.

**III . Conception détaillée**

Beaucoup de frameworks proposent une couche d'accès aux données, c'est-à-dire un système apportant bien souvent une certaine transparence vis-à-vis du Système de Gestion de Bases de Données (SGBD).

Nous n'avons plus à nous préoccuper du driver au niveau du code, puisque ce sera le rôle du framework.

Qt en fait partie et nous allons voir quelques-unes des très nombreuses possibilités qu'il offre. Bien évidemment, il existe d'autres frameworks, comme .Net par exemple, mais nous utiliserons Qt pour sa facilité d'utilisation, sa portabilité et l’intégration dans l’application LC.

Le langage utilisé sera C++ et le module QtSql.

Différents drivers sont utilisables au niveau de la connexion avec le module QtSql, mais pour nous, développeurs, nous n'avons pas à nous préoccuper de ce qui se passe à si bas niveau d'abstraction. Il est cependant utile de connaître quels drivers existent, et comment les utiliser.

Certains sont disponibles dans la version open source, d'autres nécessitent une recompilation de Qt.

Les pilotes (drivers) compatible avec le module QtSql sont répertoriés dans le tableau suivant :

Tableau -> Ligne du driver

**Exemple de programme**

Nous allons maintenant utiliser notre pilote QSQLITE.

Les connexions s'utilisent au travers de la classe QSqlDatabase.

Cette classe possède une méthode statique QSqlSatabase::addDatabase(constQString) renvoyant une instance de QSqlDatabase et reçoit en paramètre une chaîne de caractères correspondant au driver utilisé donc dans notre cas : QSQLITE.

Voici un petit programme simple pour créer une base de données, illustrant ce principe de création et connexion a une base de données :

Il y a deux différences fondamentales dans les requêtes SQL, celles qui renvoient des données, et celles qui n'en revoient pas.

La différence au niveau applicatif est qu'il faudra (ou pas) gérer un retour de données.

Dans certains frameworks, il existe plusieurs fonctions permettant d'exécuter des requêtes, certaines pour gérer des retours, d'autres non ... Cette utilisation peut paraître peu pratique aux yeux de certains. En effet, le programmeur peut se tromper de fonction et provoquer une erreur, alors que la requête est correcte.

Qt procède différemment dans le sens où il n'existe qu'un seul moyen d'envoyer la requête. Libre à nous, si on souhaite récupérer les valeurs de retour,on utilise pour cela une instance de la classe QSqlQuery en appelant sa méthode QSqlQuery::exec(constQString).

Voici le formalisme à utiliser pour envoyer des requêtes au format SQL au moteur de la base de données :

**Petit ligne de code Requete SANS retour**

Comme expliqué dans la partie précédente, la manière de requêter sera la même.

Nous avons seulement à parcourir le retour de la requête stocké au sein de l’instance QSqlQuery afin de récupérer et traiter une à une les lignes retournés par la requête.

Pour avoir accès aux données reçues, on utilise la méthode QSqlQuery::value(int) qui nous permet d’identifier la colonne (champ de la table) avec un son numéro en commençant par la gauche que l’on souhaite traiter.

Récupérer ces données nécessite de connaître parfaitement la construction de la table sélectionnée pour identifier les colonnes correctement.

Voici un exemple de l’utilisation de la méthode QSqlQuery::value(int) pour récupérer les données.

**Petit ligne de code Requete AVEC retour**

Si c'est son premier appel la méthode QSqlQuery::next() place la lecture au premier enregistrement retourné, sinon elle positionnera la lecture à l'enregistrement suivant.

*Remarque :* Il peut sembler incohérent d’utiliser des fonctions de conversions comme « toInt() » ou« toString() »car la base de données contient des données déjà formatées mais QSqlQuery::value(int) retourne un QVariant. Il s’agit en interne d’une structure proche de l’union pour les types de données de Qt les plus communs. On peut alors l’associer à n’importe quel type, ce qui semble logique car QSqlQuery::value(int) ne connais pas le type de données qu’elle traite. Nous allons donc utiliser des fonctions de conversion pour stocker les résultats de la requête renvoyée par QSqlQuery::value(int).

**Conception de la classe CSQLite\_Local\_DB**

La conception de la classe CSQLite\_Local\_DBest basée sur les principes énoncés précédemment. Les principales méthodes de la classe sont destinées à communiquer directement avec la base de données. Celle-ci sera développée avec l’environnement Qt-creator en C++.

Nous utilisons un outil de génération de documentation automatique nommé « Doxygen ». La totalité de la description de la classe est représenté à l’adresse suivante :

Afin de faciliter l’évolution et la future intégration de la classe dans l’application LC. Nous avons utilisé un gestionnaire de version : Git. Le dépôt local de git est ensuite associé à une interface web, GuiHub qui permet à chacun de travailler sur le projet.

**Test Unitaire**

Un programme de test de la classe CSQLite\_Local\_DB est disponible (unittest\_csqlite\_local\_db), écrit en c++ sous Qt avec la librairie QTestLib.

Configuré avec un gestionnaire de version Git en dépôt local et sur internet avec GitHub. Le programme peut fonctionner en deux modes possibles. Un mode manuel, en passant en paramètre le nom de la fonction de la classe CSQLite\_Local\_DB à tester et demande régulièrement à l’opérateur de continuer lui laissant le temps de vérifier la base de données par exemple. Enfin un mode automatique qui exécute toutes les fonctions de tests sans pause.

En lien avec GitHub, Travis-CI est un serveur de test d’intégration continue, lors d’un nouveau dépôt sur GuitHub, Travis-CI est averti est lance la procédure de tests automatique celui-ci indique alors l’état de l’exécution avec une image qui peut être utilisé sur le projet sur GitHub pour indiquer ou non une erreur.

Fonction membre de la classe CSQLite\_Local\_DB.

Paramètres : QList<structstruct\_PackagesUseBox> UpdatedPackagesUseBoxList

Prototype complet :

boolCSQLite\_Local\_DB::UpdateLocalData(QList<structstruct\_PackagesUseBox>UpdatedPackagesUseBoxList);

Retour : Booléen (Succès ou échec).

La fonction UpdateLocalData() met à jour la base de données locale avec les informations envoyées au serveur LMS lors de la livraison avec comme paramètre une liste des informations à enregistrer (la liste des colis utilisant les consignes et les codes de retrait associées). Retourne un booléen, à vrai (true) si la mise à jour s'est effectuée, faux (false) sinon ou si la base de données n'est pas ouverte.

**Conclusion personnelle**

|  |  |  |
| --- | --- | --- |
| **Terminé et fonctionnel** | **Non fonctionnel** | **Difficultés rencontrées** |
| Conception de la base de données |  | Etablissement des règles de gestion, modèles de données |
| classe CSQLite\_Local\_DB | Chargement d’une sauvegarde |  |
| Tests de portes | Etats réels des portes |  |
| Activation désactivation des portes | Etats réels des portes |  |
| Configuration du site | Installation des consignes | Contraintes de temps |