**BOURAOUI Rahma**

**CHAUDOIS Damien**

**4 AL 1 Groupe 16**

Rapport de projet Design Patterns

Table des matières

[Introduction 3](#_Toc517257703)

[Le carnet d’adresse 4](#_Toc517257704)

[Le code 5](#_Toc517257705)

[Les patterns 6](#_Toc517257706)

[Le singleton 6](#_Toc517257707)

[Le Composite 7](#_Toc517257708)

[Le prototype 8](#_Toc517257709)

[L’observer 9](#_Toc517257710)

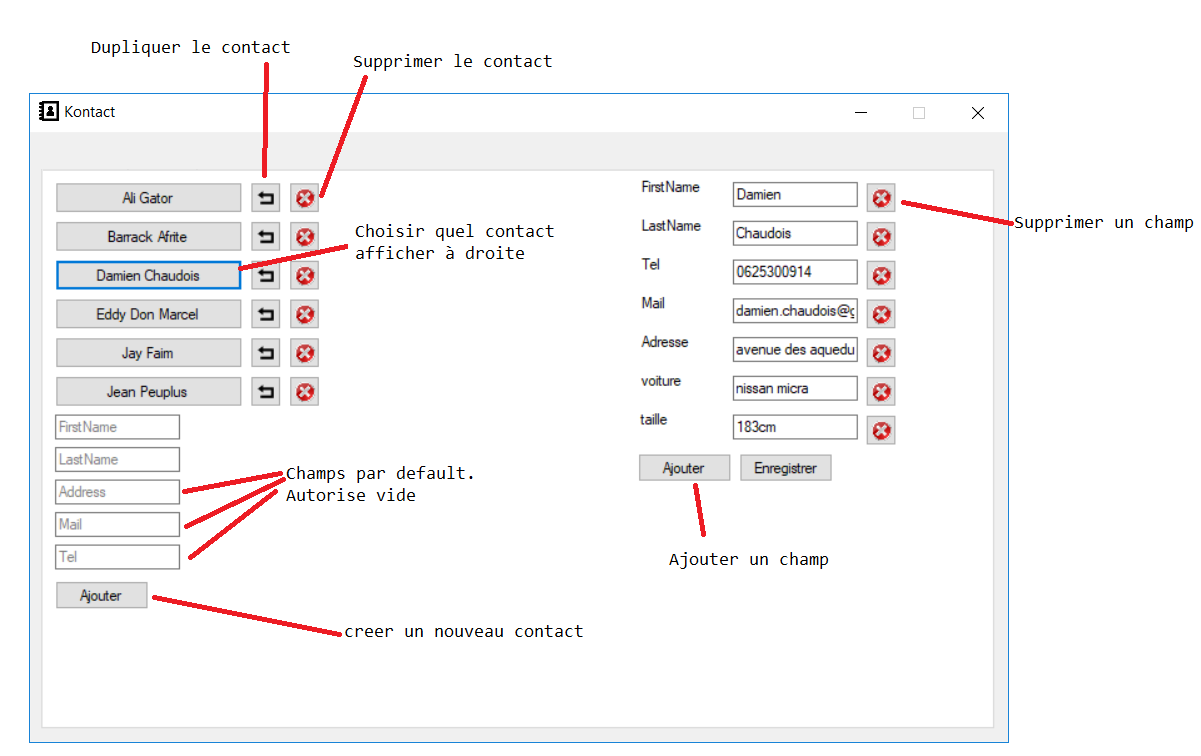
[L’itérateur 10](#_Toc517257711)

# Introduction

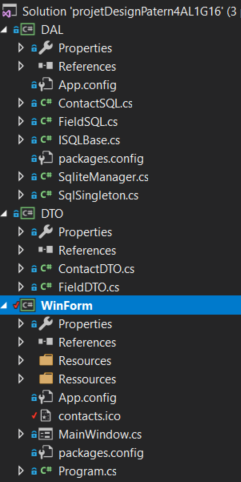
Ce rapport décrit les différents patterns qui ont été implémenter dans le projet de design pattern. Je décrirais d’abord rapidement le projet et ses objectifs, puis j’exposerais le découpage du code, et enfin je listerais les différents patterns vus en cours qui ont été utilisé pour ce projet.

# Le carnet d’adresse

Il s’agit d’une application WinForm C# qui as pour but de proposer à l’utilisateur l’enregistrement de son carnet d’adresse et de toutes les informations potentiellement utiles qui sont susceptible d’y être attaché. L’application stock les entré dans une base Sqlite3 et autorise la création d’autant de contact que désiré, et autant d’informations par contact que voulu.



# Le code

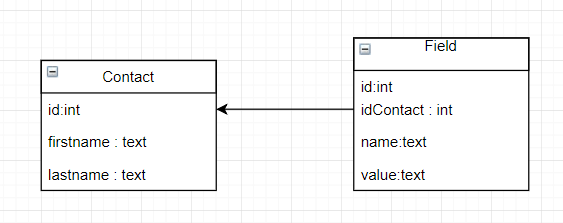


Il se découpe en 3 partie :

La vue dans le projet WinForm

Les models dans le projet DTO (Data transfert Object)

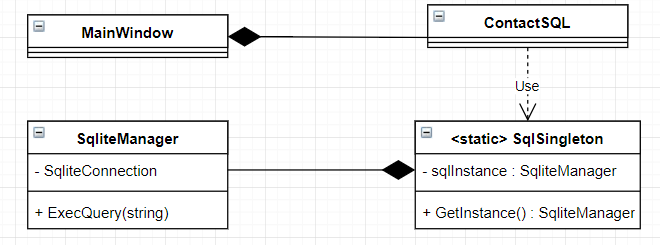
Et l’accès aux données dans la DAL (Data Access Layout)

La base de données SQLite contient deux tables :

Et sont représenté par deux DTO et deux objet SQL qui permettent d’accéder à leurs données. Lors du chargement de la fenêtre, celle-ci est remplie avec autant de ligne de contact qu’il y a de contact dans la base. Toutes les requêtes sont gérées depuis une unique instance d’une classe, créé au chargement de l’application. Au passage dans son constructeur, si la base n’existe pas, elle est créée.

# Les patterns

## Le singleton



Pour éviter les connexions multiples à la même base, il n’est possible d’accéder à la base SQLite qu’as travers le singleton :

namespace DAL

{

public static class SqlSingleton

{

private static SqliteManager sqlInstance = new SqliteManager();

public static SqliteManager getInstance()

{

return sqlInstance;

}

}

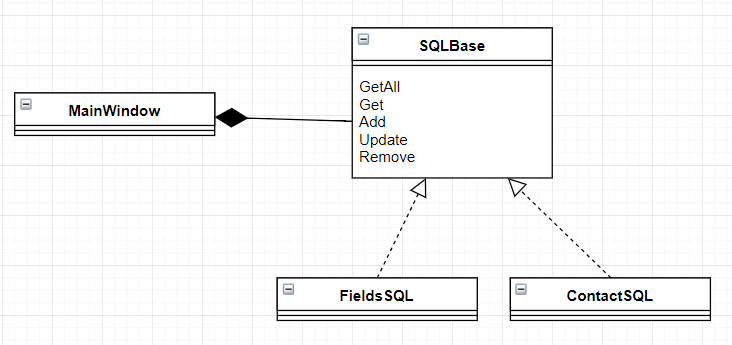
}

Après quoi pour effectuer une requête sur la base il suffit d’appeler celui-ci :

SqlSingleton.getInstance().ExecQuery("select \* from contact ;");

Toutes les DAL on ainsi accès à la même instance de SqliteManager, et ce dernier n’est fermé qu’une seul fois quand l’application se termine.

## Le Composite



L’accès à la base se fait au travers des objet d’accès : ContactSQL et FieldSQL. Tous deux implémentent « ISQLBase », qui possède toutes les fonctions pour CRUD un élément.

interface ISQLBase<T> where T: class

{

IEnumerable<T> GetAll();

T Get(int id);

void Add(T entity);

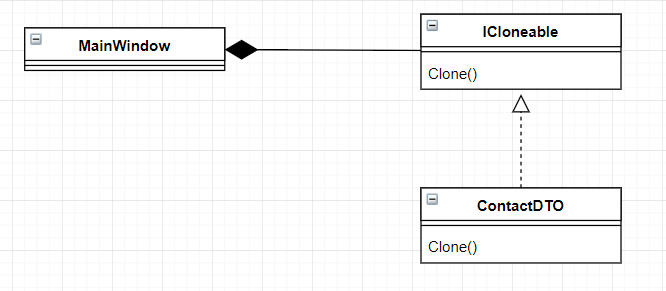
void update(T entity);

void remove(T entity);

}

Ainsi, il est garanti que tous les objets qui implémentent cette interface propose un CRUD complet sur celui-ci. Pour le développeur, il sait donc que tout les objets potentiel à venir qui implémente cette interface se comporteront de la même façon.

## Le prototype



L’application permet à l’utilisateur de dupliquer des contacts. Pour cela, l’objet ContactDTO implémente l’interface ICloneable qui lui permet de retourner un clone de son instance.

public class ContactDTO : ICloneable, IEnumerator

{

public object Clone()

{

return MemberwiseClone();

}

etc…

Le code du bouton « Dupliquer » est donc très simple :

private void CloneContact(ContactDTO item)

{

new ContactSQL().Add((ContactDTO) item.Clone());

mainPanel.Controls.Clear();

DisplayMain();

}

## L’observer

Le texte du bouton est le nom et le prénom du contact auquel il est lié. Si on modifie cette information, en principe il faudrait recharger toute la fenêtre pour mètre à jour le bouton. Ce problème est résolu grâce à l’observer.

boutonName.Click += (e, s) =>

{

DisplayInfo(item.id, boutonName);

};

Lors de la création du bouton, on lui ajoute une fonction à exécuter lors du click. En lui passant le bouton lui-même on permet à la fonction DisplayInfo de travailler sur cet objet :

private void DisplayInfo(int idcontact,Button origin)

{

boutonUpdate.Click += (e, s) =>

{

ContactDTO newContact = new ContactDTO

{

id = contact.id

};

if (origin != null)

{

origin.Text = newContact.firstName + " " + newContact.lastName;

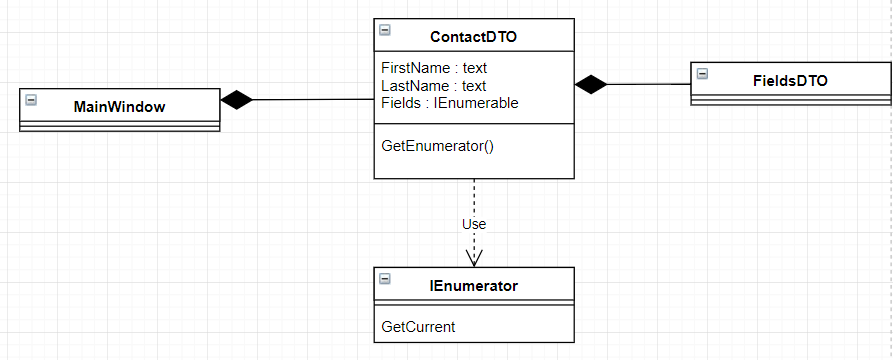
}

}

}

Ainsi la fonction DisplayInfo, au moment de créer le bouton qui enregistre ls modifications, lui ajoute un évènement sur le click qui va directement mettre à jour le bouton passé en paramètre.

## L’itérateur



La table Contact possède deux champs texte de base « nom » et « prénom » qui sont affiché sur le bouton du panneau central. Mais sur le panneau secondaire, ces données sont considérées comme des champs au même titre que ceux contenu dans la table Fields. Pour boucler dessus, il faut donc avoir un comportement personnalisé qui va poser le nom et prénom dans la même liste que les champs d’information. Pour cela, nous avont fait appel au patern iterateur en implementant l’interface IEnumerator dans l’objet ContactDTO, pour que l’on puisse l’inserer dans une boucle de type foreach :

foreach (FieldDTO field in contact)

{

textBoxInfo.Add(field.name,SetupTextBoxInfoPanel(field));

}

Dans la classe ContactDTO une variable suit l’état du parcours de l’itérateur. L’implémentation de la fonction getCurrent() se fait comme suit :

private object getCurrent()

{

if (cpt == 0)

{

return new FieldDTO

{

idContact = id,

name = "FirstName",

value = firstName

};

}

else if (cpt == 1)

{

return new FieldDTO

{

idContact = id,

name = "LastName",

value = lastName

};

}else if (cpt - 2 < fields.Count())

{

return fields[cpt - 2];

}

return null;

}