

Travaux pratiques

Sans être une fin en soi, les designs patterns guident le développeur dans la mise au point de son programme en lui évitant de réinventer la roue. L'exemple qui suit illustre le principe de séparation du code par niveau de préoccupation (on parle de "couches" de code) et s'appuie sur le design pattern **Fabrique** (*Factory*).

1. L'objectif du programme

Une classe de donnée assez simple, `Livre` contient deux attributs `auteur` et `titre`.

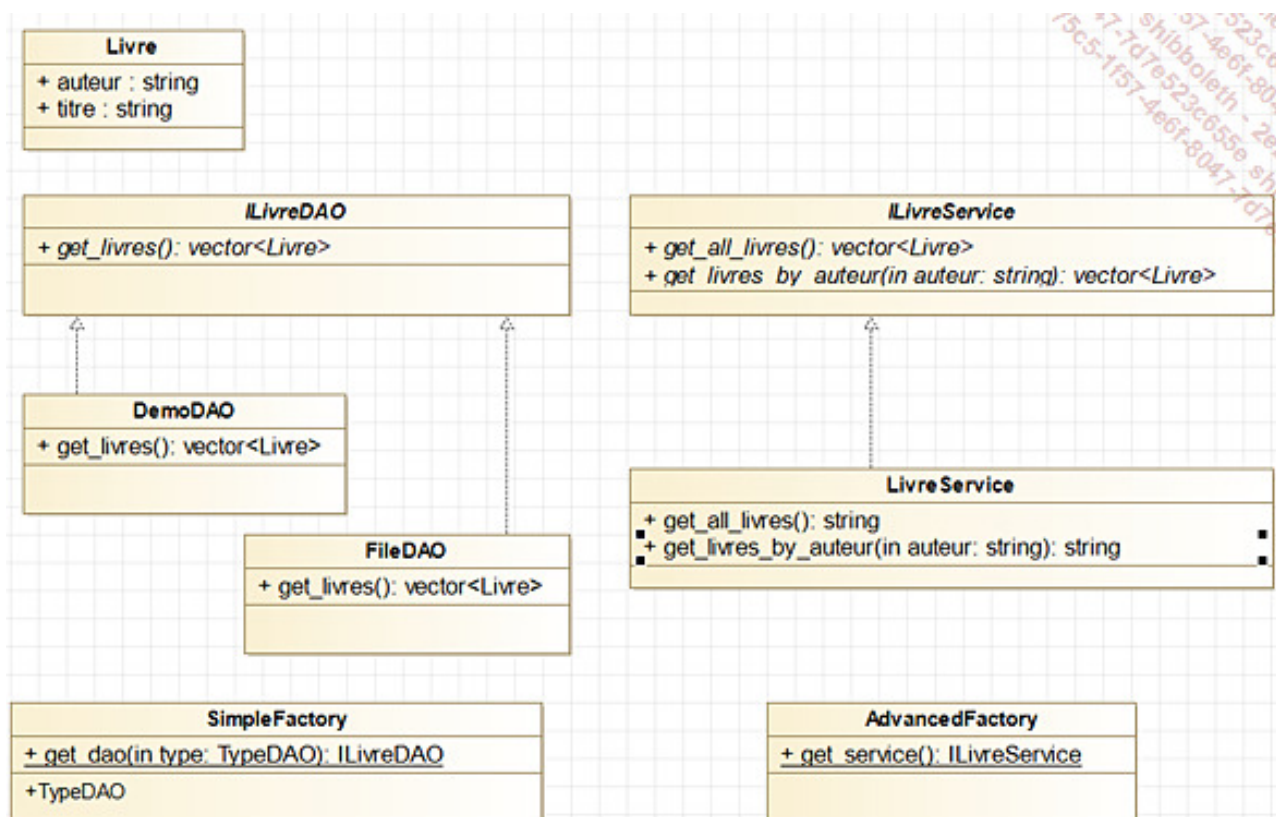
Le programme comporte une classe de **service** afin de lister des livres à partir de différentes sources (appelées **DAO** pour *data access object*) telles qu'une base de données, un fichier ou bien une sélection prédéfinie (autrement dit une liste codée en dur).

La fonction `main()` réalise deux appels à la classe de service, le premier pour lister tous les livres du catalogue, le second pour lister les ouvrages filtrés par auteur.

2. Le diagramme de classes

Certaines classes sont assorties d'une classe d'interface qui définit le contrat fonctionnel au travers d'une liste de méthodes virtuelles pures (aussi appelées méthodes abstraites en C# ou en Java).

Cette construction est assez classique dans le découpage d'un programme en couches, et elle est d'autant plus utile que le design pattern **Factory** entre en jeu.



Les classes d'interfaces sont représentées comme des classes abstraites puisque leurs méthodes ne comprennent pas d'implémentations (ce sont des méthodes virtuelles pures).

En C++, voici comment s'écrit cette association classe d'interface - classe :

```

/*
  classe d'interface ILivreDAO
  Ne contient que des méthodes virtuelles pures (sans code)
  Ne peut être instanciée telle quelle sans implémentation
*/
class ILivreDAO
{
public:
  virtual vector<Livre> get_livres() = 0;
};

/*
  DemoDAO implémente ILivreDAO pour lister des livres selon

```

```

une sélection prédéfinie
*/
class DemoDAO :
    public ILivreDAO
{
public:

    // implémentation de la méthode virtuelle définie
    dans la classe d'interface
    virtual vector<Livre> get_livres();
};

```

Nous indiquons ici le code de la méthode `DemoDAO::get_livres()` pour souligner son caractère non abstrait, autrement la classe `DemoDAO` ne serait pas instanciable :

```

vector<Livre> DemoDAO::get_livres() {

    vector<Livre> livres;

    livres.push_back(Livre("Victor Hugo", "Les misérables"));
    livres.push_back(Livre("Victor Hugo", "Notre-Dame de Paris"));
    livres.push_back(Livre("Victor Hugo", "Ruy Blas"));
    livres.push_back(Livre("Gustave Flaubert", "Madame Bovary"));
    livres.push_back(Livre("Edmond Cervantes", "Don Quichotte"));

    return livres;
}

```

3. La fabrique SimpleFactory

La classe `SimpleFactory` réalise le design pattern Factory ; son rôle est d'instancier une classe selon un schéma qui peut dépendre de telle ou telle configuration.

```

/*

```

```

    SimpleFactory réalise le design pattern "Factory"
    Instancie une implémentation de la classe d'interface
    ILivreDAO selon le paramètre TypeDAO
    */
class SimpleFactory
{
public:
    enum class TypeDAO { DAO_DEMO, DAO_FILE };
    static ILivreDAO* get_dao(TypeDAO type);
};

```

L'implémentation de la méthode `get_dao()` est assez directe, pourtant elle garantit au programme un certain niveau d'abstraction et donc de stabilité vis-à-vis des changements de configuration :

```

ILivreDAO* SimpleFactory::get_dao(TypeDAO type) {
    if (type == TypeDAO::DAO_DEMO)
        return new DemoDAO();

    if (type == TypeDAO::DAO_FILE)
        return new FileDAO();

    return nullptr;
}

```

Passons à la classe `LivreService`, elle s'appuie sur la fabrique d'objets pour récupérer un DAO selon la configuration idoine :

```

/*
    LivreService implémente la classe d'interface ILivreService
    */
class LivreService :
public ILivreService
{
public:
    // utilisé par les méthodes de service
    ILivreDAO* dao;
}

```

```

// constructeur par défaut
// instancie dao à l'aide de SimpleFactory
LivreService() {
    dao = SimpleFactory::get_dao( SimpleFactory::TypeDAO::DAO_DEMO);
}

// Hérité via ILivreService
virtual vector<Livre> get_all_livres() override;
virtual vector<Livre> get_livres_by_auteur(string auteur) override;
};

```

Dans la méthode `main()` on passe par la couche service pour récupérer la liste complète de livres :

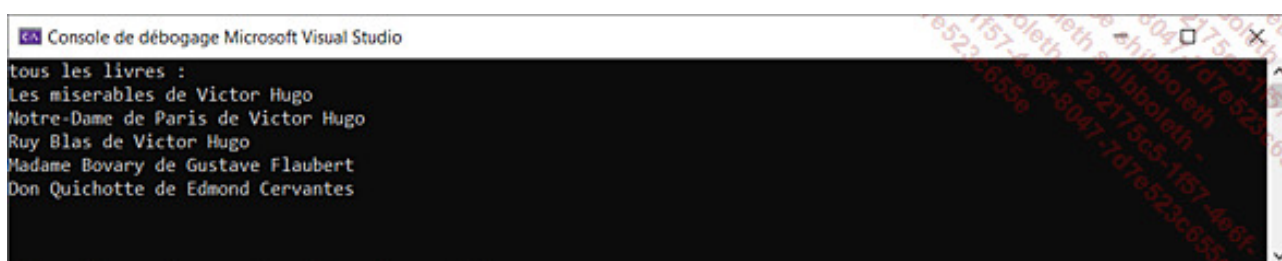
```

int main()
{
    ILivreService* service(new LivreService());

    cout << "tous les livres :" << endl;
    auto all = service->get_all_livres();
    for (auto it = all.begin(); it < all.end(); it++)
        cout << (*it).titre << " de " << (*it).auteur << endl;

    cout << endl;
}

```



```

Microsoft Visual Studio
tous les livres :
Les misérables de Victor Hugo
Notre-Dame de Paris de Victor Hugo
Ruy Blas de Victor Hugo
Madame Bovary de Gustave Flaubert
Don Quichotte de Edmond Cervantes

```

4. L'injection de dépendances

L'injection de dépendances est une technique d'initialisation en cascade d'objets. Le constructeur d'une classe reçoit en paramètre les objets qui serviront aux méthodes sans avoir à les initialiser lui-même. Cela permet notamment de centraliser la configuration et d'éviter de propager du contexte à toutes les classes.

La classe `AdvancedFactory` utilise justement cette technique pour instancier le DAO dans la couche service.

Pour cela, un deuxième constructeur est défini dans la classe `LibreService` avec un DAO comme paramètre :

```
/*
  LibreService implémente la classe d'interface ILivreService
*/
class LibreService :
public ILivreService
{
public:
  // utilisé par les méthodes de service
  ILivreDAO* dao;

  // constructeur par défaut
  // instancie dao à l'aide de SimpleFactory
  LibreService() {
    dao =
SimpleFactory::get_dao(SimpleFactory::TypeDAO::DAO_DEMO);
  }

  // constructeur avec injection de dépendance
  // reçoit une instance de ILivreDAO
  LibreService(ILivreDAO* dao) : dao(dao) {

  }

  // Hérité via ILivreService
  virtual vector<Livres> get_all_livres() override;
  virtual vector<Livres> get_livres_by_auteur(string auteur) override;
};
```

La méthode `AdvancedFactory::get_service()` réalise l'instanciation du DAO

puis l'**injection de dépendances** :

```
ILivreService* AdvancedFactory::get_service()
{
    // instanciation de la dépendance à l'aide de SimpleFactory
    ILivreDAO* dao = SimpleFactory::get_dao(SimpleFactory::TypeDAO::DAO_DEMO);

    // injection de la dépendance
    return new LivreService(dao);
}
```

La méthode `main()` ne révèle pas ce qui se passe en coulisse :

```
ILivreService* service_di = AdvancedFactory::get_service();
string auteur = "Victor Hugo";
cout << "tous les livres de " << auteur << " : " << endl;
auto all_auteur = service_di->get_livres_by_auteur(auteur);
for (auto it = all_auteur.begin(); it < all_auteur.end(); it++)
    cout << (*it).titre << endl;
```



```
Console de débogage Microsoft Visual Studio
tous les livres de Victor Hugo :
Les misérables
Notre-Dame de Paris
Ruy Blas

Sortie de C:\Users\gueri\source\repos\factory_di\Debug\factory_di.exe (processus 22268). Code : 0.
Appuyez sur une touche pour fermer cette fenêtre. . .
```