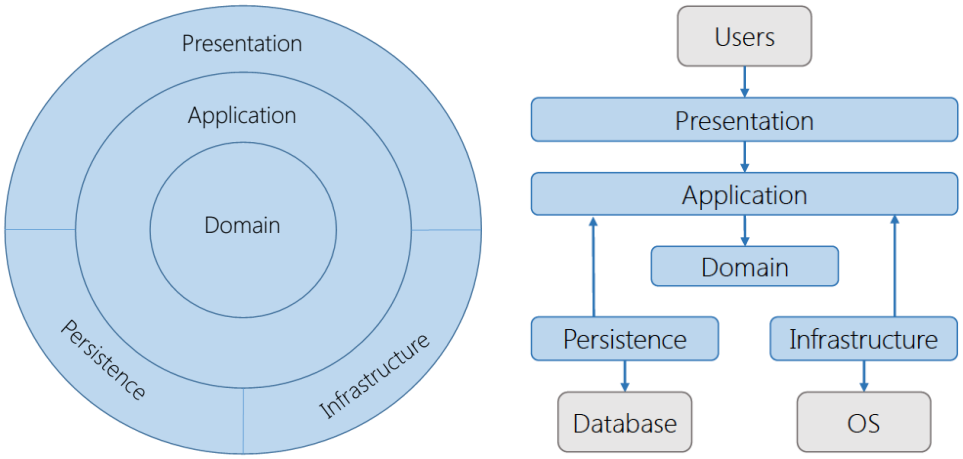
Travaux dirigés 4

Persistance des objets

Les objets passent par une série d'états au cours de leur vie. Ils sont créés, placés en mémoire et utilisés dans des traitements, puis ils sont détruits. Dans certains cas, ils sont sauvegardés dans des emplacements permanents, comme une base de données ou un archive, afin de les récupérer plus tard. A un moment donné, ils peuvent être complètement effacés du système, y compris de la base de données ou de l'archive.

Il existe désormais un grand nombre de solutions pour gérer la persistance des objets. Cependant, une conception médiocre serait de coupler le système à une solution particulière, en éparpillant un peu partout du code très spécifique pour récupérer les données et les transformer en objet. C'est pourquoi il est important de prévoir une couche d'abstraction dédiée à l'accès aux données. Celle-ci doit créer l'illusion d'une base de données orientée objet en définissant les correspondances entre les objets persistants du système et la solution de stockage choisie (base de données relationnelle, fichier XML, etc). Il en résulte ainsi un code organisé en couches bien séparées (voir la figure).



(a) Architecture centrée sur le domaine.

(b) Séparation en quatre couches.

Le présent TD est organisé en trois sections, dont les deux premières proposent des exercices qui visent à faire comprendre les principes de conception pour une couche d'accès aux données, tandis que la troisième section porte sur un mini-projet concernant la réalisation d'un logiciel de gestion.

4.1 Patrons d'accès aux données

La manière la plus intuitive pour manipuler une base de données consiste à représenter chaque ligne d'une table relationnelle par un objet, qui regroupe les méthodes d'accès aux données, ainsi que les méthodes applicatives utilisant ces données. Nous avons donc la correspondance suivante :

1. une table dans la base → une classe,
2. une ligne dans la table → un objet,
3. une colonne dans la table → un attribut.

Ce faisant, l'accès aux données passe par les méthodes de création, lecture, mise à jour et suppression, qui définissent l'interface CRUD. Voici un exemple.

```
public class Post
{
    public long     Id;
    public string   Text;
    public DateTime Date;

    public void Create() {
        ... // SQL INSERT
    }
    public static Post Read(long id) {
        ... // SQL SELECT
    }
    public void Update() {
        ... // SQL UPDATE
    }
    public void Delete() {
        ... // SQL DELETE
    }
}
```

Aux fins de la persistance, il faut d'abord mettre en correspondance la classe ci-dessus avec une table relationnelle dans la base de données.

```
string table = "CREATE TABLE Post("
             + "id   INTEGER PRIMARY KEY,"
             + "text TEXT    NOT NULL,"
             + "date INTEGER NOT NULL)";
```

La méthode `Create()` insère un nouvel objet dans la base de données, qui se charge de générer un identifiant unique pour l'objet en question.

```
string insert = "INSERT INTO Post(text,date) VALUES ('{0}', '{1}')";
```

La méthode `Read()` retrouve un objet précis dans la base de données.

```
string select = "SELECT * FROM Post WHERE id = {0}";
```

La méthode `Update()` modifie un objet déjà présent dans la base de données.

```
string update= "UPDATE Post SET text='{0}',date='{1}' WHERE id={2}";
```

La méthode `Delete()` supprime un objet contenu dans la base de données.

```
string delete = "DELETE FROM Post WHERE id={0}";
```

4.1.1 Entrepôt

Bien que l'approche précédente soit simple et intuitive, elle présent l'inconvénient de mélanger des fonctions hétéroclites au sein du même objet. A cet égard, le patron *entrepôt* propose de déplacer la logique d'accès aux données dans un objet séparé, qui agit comme une sorte d'entrepôt en mémoire des instances emmagasinées dans la base de données. Cet entrepôt expose une interface CRUD qui dépende uniquement d'une structure de données regroupant les attributs persistants. Voici un exemple.

```
class Post
{
    public long Id;
    public string Text;
    public DateTime Date;
}
```

```
interface IRepository<DTO>
{
    void Create(DTO entity);
    DTO  Read  (long Id);
    void Update(DTO entity);
    void Delete(DTO entity);
    IQueryable<DTO> Query();
}
```

L'avantage de cette approche est que la mise en correspondance objet-relationnel est concrètement implémentée dans une classe à part.

4.1.2 Unité de travail

Une erreur courante lors de l'accès aux données est d'utiliser les transactions uniquement en cas d'écritures multiples. En réalité, tout opération d'accès doit être effectuée dans une transaction, y compris les lectures, afin d'obtenir des données cohérentes et non corrompues par des écritures partielles. Dans l'implémentation de l'entrepôt présentée dans la section précédente, chaque accès à la base de données est effectué dans une transaction différente crée implicitement. Le patron *unité de travail* propose de déplacer la gestion des transactions dans un objet séparé et distinct de l'entrepôt, permettant ainsi d'organiser les accès aux données en différentes unités de travail, chacune exécutée dans sa propre transaction.

L'objet qui encapsule l'unité de travail est définie par une interface ayant les méthodes `Commit()` et `Dispose()`, lesquelles permettent de faire le *commit* ou le *rollback* d'une transaction. Pour garantir la complète isolation du reste du système, l'unité de travail et l'entrepôt sont instanciés par un objet « fabrique », dont l'interface `IDataAccess` est définie ci-après.

```
public interface IUnitOfWork : IDisposable
{
    void Commit();
}

interface IDataAccess : IDisposable
{
    IUnitOfWork BeginTransaction();

    IRepository<T> CreateRepository<T>();
}
```

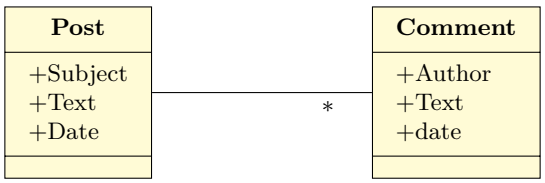
La création d'un objet `IUnitOfWork` détermine le début d'une nouvelle transaction, tandis qu'un appel à la méthode `Commit()` correspond à sa fin. Par conséquent, tous les accès aux données doivent se faire entre le début et la fin d'une transaction, par le biais des entrepôts. Voici un exemple.

```
public class Service {
    IDataAccess dataAccess;

    public Service(IDataAccess dataAccess) {
        this.dataAccess = dataAccess;
    }

    public void ModifyYourData() {
        Repository<Post> repo = dataAccess.CreateRepository<Post>();

        // Begin a new transaction
        using (IUnitOfWork uow = dataAccess.BeginTransaction())
        {
            int id = 142;
            Post post = repo.Read(id);    // Get a persistent object

            post.Text = "Modified text"; // Modify the object

            repo.Update(post);           // Update the database

            uow.Commit();                // End the transaction with a commit
        }
    }
}
```

4.1.3 Agrégat

Les objets persistants sont souvent manipulés en groupe pour assurer l'intégrité des données. Dans ce contexte, un *agrégat* est un groupe d'objets connexes qui sont considérés comme un tout unique vis-à-vis des modifications de données. L'un de ces objets assure le rôle de *racine*. Celle-ci peut posséder des références vers n'importe quel objet de l'agrégat, et les autres objets peuvent se référencer entre eux, mais un objet à l'extérieur de l'agrégat peut seulement avoir une référence vers la racine. Par conséquent, la seul manière de manipuler un agrégat est de demander à la racine d'effectuer des actions. Si celle-ci est supprimée, tous les objets de l'agrégat seront supprimés aussi.

Un exemple d'agrégat est illustré dans la suite. Cet agrégat regroupe les objets de type `Post` et `Comment`, qui sont liés entre eux par une relation *un-à-plusieurs*, où le rôle da racine est joué par les objets de type `Post`.



4.2 Fluent NHibernate

La librairie *Fluent NHibernate* automatise la mise en correspondance entre le modèle objet et le modèle relationnel, sans se soucier du langage SQL. En particulier, cet outil permet de déduire le schéma des tables relationnelles à partir des classes définissant les objets persistants, à condition que celles-ci suivent des règles précises. Voici quelques unes des règles principales.

1. Un attribut persistant est défini par une propriété publique et virtuelle.
2. Un entier nommé `Id` avec le setteur protégé représente la clé primaire.
3. Une propriété de type simple représente une relation de un à un.
4. Une propriété de type liste représente une relation de un à plusieurs.
5. Toute méthode publique doit être déclarée virtuelle.

L'exemple suivant illustre l'implémentation des classes `Post` et `Comment`, dont les objets forment l'agrégat mentionné dans la section précédente.

```
namespace Entities
{
   public class Post
   {
      public virtual int             Id      { get; protected set; }
      public virtual string          Subject { get; set; }
      public virtual string          Text    { get; set; }
      public virtual DateTime        Date    { get; set; }
      public virtual IList<Comment> Comments{ get; set; }

      public Post() {
         Comments = new List<Comment>();
      }

      public virtual void Add(Comment comment) {
         comment.Post = this;
         Comments.Add(comment);
      }
   }

   public class Comment {
      public virtual int      Id    { get; protected set; }
      public virtual string   Text  { get; set; }
      public virtual string   Author{ get; set; }
      public virtual DateTime Date  { get; set; }
      public virtual Post     Post  { get; set; }
   }
}
```

La mise en correspondance est faite automatiquement par la commande `AutoMap.AssemblyOf<Post>()`, qui génère le schéma suivant.

```
CREATE TABLE Post
(
   Id      INT PRIMARY KEY,
   Subject TEXT,
   Text    TEXT,
   Date    DATETIME

);
```

```
CREATE TABLE Comment
(
   Id     INT PRIMARY KEY,
   Text   TEXT,
   Author TEXT,
   Date   DATETIME,
   PostId INT REFERENCES Post(Id)
);
```

Le schéma relationnel est nécessaire pour la création d'un point d'accès à la base de données. Cette tache est réalisée par le code ci-dessous, où la base de données est un fichier SQLite nommé `Data.db`, le schéma des tables relationnelles est généré à partir des classes faisant partie de l'espace de noms `Entities`, le lazy loading est activé, et les modifications sont en cascade.

```
public class ORM<T>
{
   static readonly string FileDB = "Data.db";

   public static ISessionFactory CreateSessionFactory(bool test=false) {
      return test ? CreateSessionFactory(BuildSchemaAlways)
                  : CreateSessionFactory(BuildSchemaIfFileIsMissing);
   }

   // Paramétrisation du point d'acces
   static ISessionFactory CreateSessionFactory(Action<Configuration> BuildSchema) {
      return Fluently.Configure()
                     .Database( SQLiteConfiguration.Standard.UsingFile(FileDB) )
                     .Mappings( m => m.AutoMappings.Add(CreateAutomappings) )
                     .ExposeConfiguration(BuildSchema)
                     .BuildSessionFactory();
   }

   // Mise en correspondance objet-relationnel
   static AutoPersistenceModel CreateAutomappings() {
      return AutoMap.AssemblyOf<T>()
                    .Where(type => type.Namespace == "Entities")
                    .Conventions.Add( DefaultCascade.All(), DefaultLazy.Always() );
   }

   // Création de la base de données
   static void BuildSchemaIfFileIsMissing(Configuration config) {
      if (!File.Exists(FileDB))
         new SchemaExport(config).Create(false, true);
   }
   static void BuildSchemaAlways(Configuration config) {
      if (File.Exists(FileDB))
         File.Delete(FileDB);
      new SchemaExport(config).Create(false, true);
   }
}
```

Ainsi, l'implémentation de la classe `UnitOfWork` devient extrêmement facile.

```
public class UnitOfWork : IUnitOfWork
{
   ISession session;
   ITransaction trans;

   public UnitOfWork(ISession session) {
      trans = session.BeginTransaction();
   }

   public void Commit() {
      trans.Commit();
   }

   public void Dispose() {
      trans.Dispose(); // rollback the transaction if it was not committed
   }
}
```

La classe `Repository` est simplement implémentée avec un type générique.

```
public class Repository<DTO> : IRepository<DTO>
{
    ISession session;

    public Repository(ISession session) {
        this.session = session;
    }

    public void Create(DTO entity)
    {
        session.Save(entity);
    }

    public DTO Read(int id)
    {
        return session.Get<DTO>(id);
    }

    public void Update(DTO entity)
    {
        session.Update(entity);
    }

    public void Delete(DTO entity)
    {
        session.Delete(entity);
    }

    public IQueryable<DTO> Query()
    {
        return session.Query<DTO>();
    }
}
```

La classe `DataAccess` s'occupe d'établir la connexion à la base de données.

```
public class DataAccess : IDataAccess
{
    ISession session;

    public DataAccess(ISessionFactory factory) {
        session = factory.OpenSession();
    }

    public IUnitOfWork BeginTransaction()
    {
        return new UnitOfWork(session);
    }

    public IRepository<T> CreateRepository<T>()
    {
        return new Repository<T>(session);
    }

    public void Dispose()
    {
        session.Dispose(); // Close Session
    }
}
```

Exercice 4.19

Créer une solution Visual Studio avec un projet « Unit Test » et ajouter à ceci les paquets NuGet `System.Data.SQLite.Core` et `FluentNHibernate`.

1. Importer dans le projet les interfaces `IRepository`, `IUnitOfWork`, `IDataAccess` et les classes `ORM`, `UnitOfWork`, `Repository` et `DataAccess`.
2. Créer la classe qui accueillera les tests pour l'accès à la base de données. Noter que la base de données est écrasée **au début de chaque test**.

```
[TestClass]
public class RepositoryTest
{
    ISessionFactory factory;
    IDataAccess access;
    IRepository<Post> postRepo;
    IRepository<Comment> commentRepo;

    Post post1, post2;

    [TestInitialize]
    public void SetUp()
    {
        CreateAccessPoint();
        CreateFixtures();
        PopulateDatabase();
    }

    void CreateAccessPoint() {
        factory = ORM<Post>.CreateSessionFactory(true);
        access = new DataAccess(factory);
        postRepo = access.CreateRepository<Post>();
        commentRepo = access.CreateRepository<Comment>();
    }

    void CreateFixtures() {
        DateTime date1 = new DateTime(2012, 9,  3); // 03 sep 2012
        DateTime date2 = new DateTime(2017, 6, 15); // 15 jun 2017
        post1 = new Post{ Subject = "Sport", Date = date1 };
        post1.Add(new Comment{ Author = "Tizio", Date = date1 });
        post1.Add(new Comment{ Author = "Caio", Date = date2 });
        post2 = new Post{ Subject = "Music", Date = date2 };
    }

    void PopulateDatabase()  {
        using (ISession session = factory.OpenSession())
        using (ITransaction trans = session.BeginTransaction())
        {
            session.Save(post1);
            session.Save(post2);
            trans.Commit();
        }
    }

    [TestCleanup]
    public void TearDown() {
        access.Dispose();
        factory.Dispose();
    }
}
```

3. Vérifier le bon fonctionnement de la classe `Repository` en utilisant des tests unitaires. Voici un exemple de test pour la méthode `Create()`.

```
[TestMethod]
public void WhenCreated_ObjectGetsId()
{
   Post p = new Post{ Subject="Sport", Date=DateTime.Now };

   using (IUnitOfWork uow = access.BeginTransaction())
   {
      postRepo.Create(p);
      uow.Commit();
   }

   Assert.IsTrue(p.Id > 0);
}
```

   Tester les différentes cas d'utilisation de l'interface `IRepository`, notamment la création d'un nouvel objet, la lecture d'un objet persistent (déjà présent dans la base de donnés), la modification d'un objet persistent, ainsi que la suppression d'un objet persistent. Pour mener à bien chaque test, il faut accéder deux fois à la base de données :

   — la première fois pour effectuer une écriture qui modifie les données persistantes selon la manière souhaitée ;

   — la deuxième fois pour effectuer une lecture permettant de vérifier que les données persistantes sont effectivement changées.

4. Écrire des tests unitaires pour la méthode `Query()`. Celle-ci permet d'effectuer des lectures dans la base de données sans besoin d'écrire explicitement des requêtes SQL. Voici quelques exemples d'utilisation.

   — La lecture de tous les objets `Post` présent dans la base de données (`SELECT * FROM Post`) peut être réalisé de la manière suivante.

```
IList<Post> all = postRepo.Query().ToList();
```

   — La lecture de tous les objets `Post` ayant le sujet "sport" peut être réalisé de la manière suivante.

```
IList<Post> bySubject = postRepo.Query()
                                .Where(p => p.Subject=="Sport")
                                .ToList();
```

   — La lecture de tous les objets `Post`, dont la date est comprise dans un intervalle donné, peut être réalisé de la manière suivante.

```
DateTime begin = new DateTime(2008, 2, 21); // 21 feb 2008
DateTime end   = new DateTime(2012, 9,  3); //  3 sep 2012

IList<Post> byDate = postRepo.Query()
                             .Where(c => c.Date >= begin)
                             .Where(c => c.Date <= end)
                             .ToList();
```

   Consultez la documentation de `Query()` pour plus d'informations.

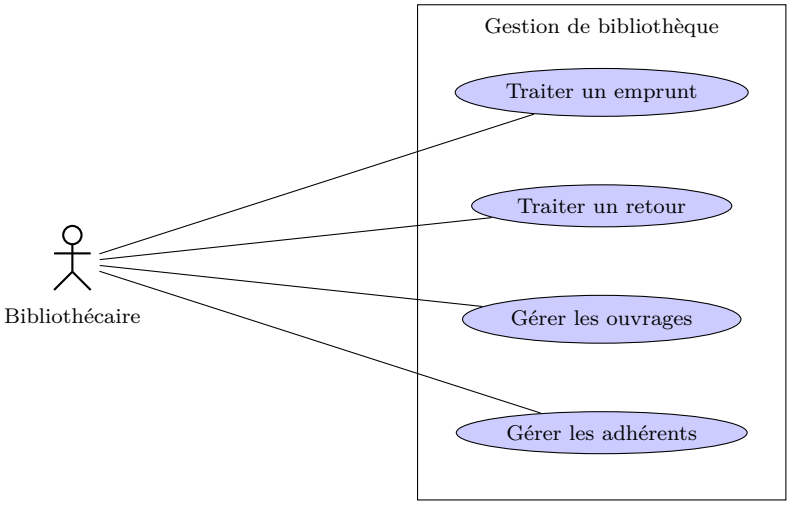
4.3 Mini-projet : gestionnaire de bibliothèque

L'objectif de cette étude est de développer un système qui informatise le référencement des ouvrages et la gestion des prêts d'une bibliothèque. Pour simplicité, les seuls ouvrages traités sont des livres. Voici une liste de besoins.

1. La bibliothécaire doit pouvoir saisir des nouveaux ouvrages, mettre à jour des ouvrages existants, et éventuellement en supprimer. Elle doit pouvoir réaliser le même type d'opérations sur les adhérents.
2. Attention à la distinction entre oeuvre et exemplaire. Une bibliothèque possède généralement plusieurs exemplaires d'une même oeuvre, et ce sont ces derniers qui sont empruntés.
3. La bibliothécaire doit pouvoir gérer les prêts, notamment l'emprunt et le retour des exemplaires aux adhérents.
4. Un adhérent ne peut pas emprunter plus que cinq ouvrages.
5. La bibliothécaire doit pouvoir effectuer des recherches pour savoir si un ouvrage existe, s'il est disponible pour le prêt, et qui à emprunté les exemplaires en cours de prêt.

4.3.1 Description des besoins

L'étude d'un logiciel doit toujours commencer par l'identification des acteurs principaux et de leurs buts. En l'occurrence, le cahier des charges dit que la bibliothécaire est le seul acteur à interagir avec le système, dans le but de gérer les prêts, les adhérents et les ouvrages. Ces considérations permettent de tracer le diagramme de cas d'utilisation suivant.



Le diagramme de cas d'utilisation fournit une représentation graphique succincte du contexte de l'application, en présentant les acteurs et la façon dont ils utilisent le système. La finalité des cas d'utilisation est de centrer l'expression des besoins sur les utilisateurs, en définissant le comportement du système de leur point de vue. Un cas d'utilisation est avant tout une description textuelle. Il raconte sous forme de texte la façon dont le système est utilisé pour permettre à un acteur d'atteindre ses objectifs et d'obtenir un résultat observable qui lui apporte de la valeur. Modéliser les cas d'utilisation signifie avant tout rédiger. Si un équipe passe des heures sur un diagramme de cas d'utilisation au lieu de se consacrer à la rédaction, elle se trompe de priorité et gaspille ses efforts. Vous trouvez ci-dessous le texte du cas d'utilisation « *traiter un emprunt* ». Inspirez-vous de cette exemple pour rediger le cas d'utilisation « *traiter un retour* ».

Traiter un emprunt

Point d'entrée. Un adhérent souhaite emprunter un ouvrage.

Scénario principal.

1. Le système visualise les opérations disponibles à la bibliothécaire (elle est déjà identifiée et authentifiée par le système).
2. La bibliothécaire commence un nouvel emprunt à l'adhérent.
3. Le système vérifie que l'adhérent peut emprunter des ouvrages.
4. La bibliothécaire indique l'exemplaire à emprunter.
5. Le système vérifie que l'exemplaire est éligible pour l'emprunt.
6. La bibliothécaire termine l'emprunt.
7. Le système enregistre les informations relatives au prêt, à savoir la date d'emprunt, l'exemplaire emprunté et l'adhérent concerné.

Scénarios alternatifs.

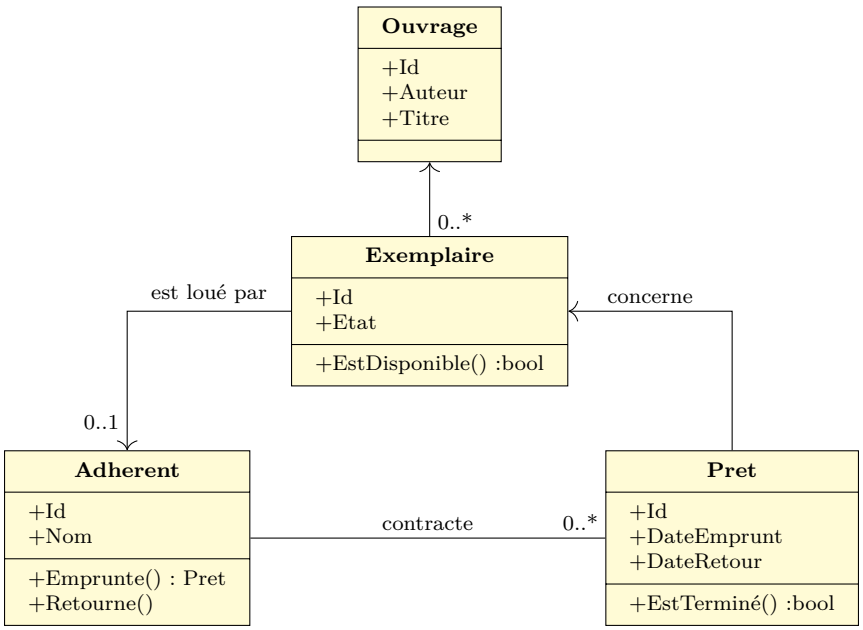
3.1 L'identifiant de l'adhérent n'est pas reconnu :
   (a) Le système prévient la bibliothécaire.
   (b) Le cas d'utilisation se termine en échec.

3.2 L'adhérent a dépassé le quota sur les prêts :
   (a) Le système prévient la bibliothécaire et termine l'emprunt.
   (b) Le cas d'utilisation se termine en échec.

5.1 L'exemplaire est en cours de prêt :
   (a) Le système prévient la bibliothécaire.
   (b) Le scénario principal reprend au point 4.

4.3.2 Modélisation du domaine

Une fois que les cas d'utilisation principaux sont identifiés, on peut passer à l'étude du domaine. En lisant le cahier des charges, on déduit que les concepts importants sont les **adhérents**, les **ouvrages**, les **exemplaires** et les **prêts**, lesquels sont reliés de la manière suivante.



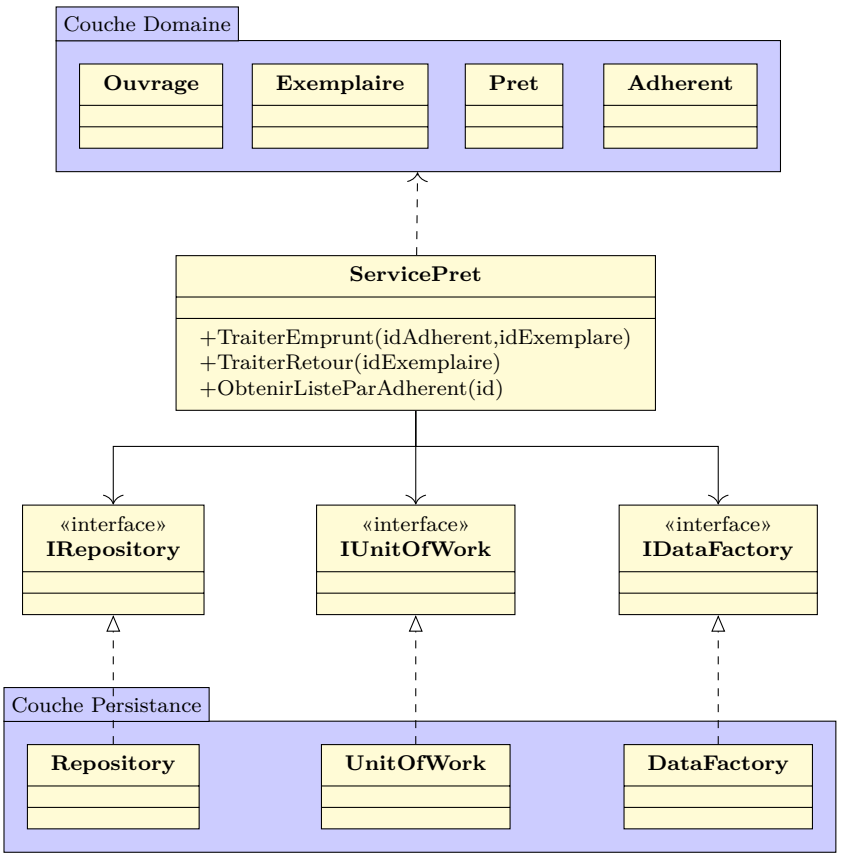
Les classes ci-dessus forment un agrégat dont `Adherent` est la racine. Celle-ci possède deux méthodes pour traiter les emprunts et les retours. Toute manipulation de l'agrégat doit passer par ces deux méthodes, afin d'assurer la cohérence des relations et l'intégrité des données. Voici leurs spécifications.

1. `Emprunte()`. Elle prend en entrée un objet `Exemplaire`. Si ceci n'est pas emprunté, un objet `Pret` est créé, sa date d'emprunt est enregistrée, et ses liens avec les objets `Exemplaire` et `Adherent` sont établis.
2. `Retourne()`. Elle prend en entrée un objet `Exemplaire`. Si ceci est lié à un objet `Pret` se trouvant dans la liste des prêts de l'adhérent, la date de retour est enregistrée, et le lien avec l'objet `Adherent` est nullifié.

INSTRUCTIONS Pour réaliser le gestionnaire de bibliothèque, nous vous fournissons une solution Visual Studio contenant une simple interface utilisateur (dossier `IHM`), les classes pour accéder aux données persistantes (dossier `Persistance`), et la définition vide des classes susmentionnées (dossier `Domaine`). Votre tache est de compléter l'implémentation de ces classes. Vous êtes autorisés à ajouter d'autres méthodes ou attributs si nécessaire. Également, vous êtes encouragés à utiliser le développement piloté par les tests (un projet « Unit Test » est déjà inclus dans la solution Visual Studio).

4.3.3 Conception de la couche applicative

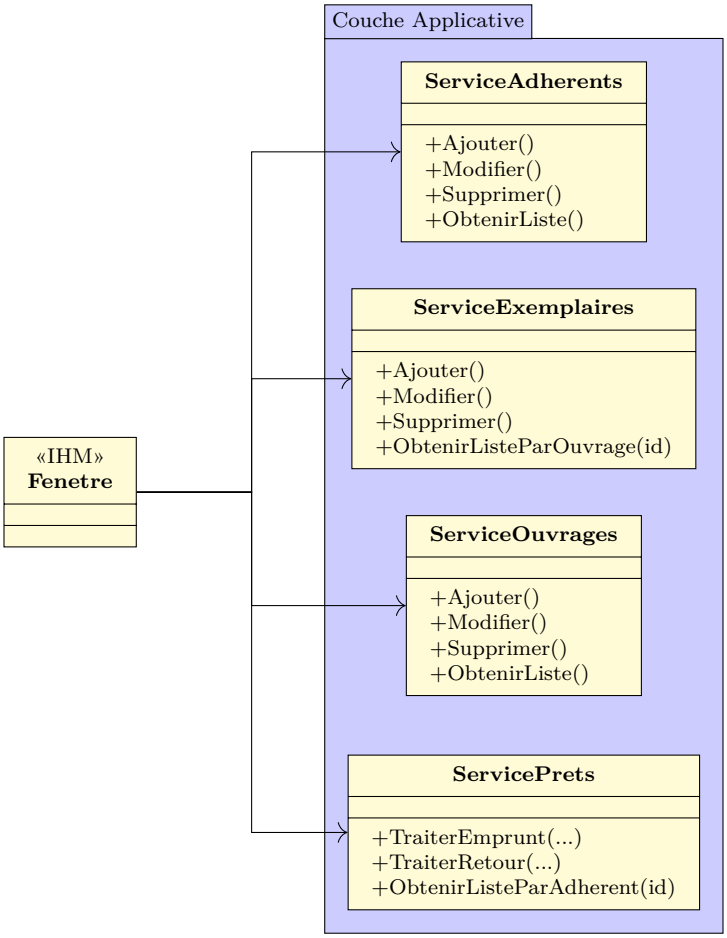
Les classes du domaine sont placées au coeur du système, dans la couche `Domaine`. Celle-ci est indépendante de toute autre partie du système, et c'est pourquoi elle est bâtie en premier. L'étape suivante est normalement la conception de la couche applicative. Il s'agit d'une couche fine qui coordonne l'activité des objets du domaine. En l'occurrence, elle contient la logique pour réaliser les cas d'utilisation, mais elle délègue les opérations métier aux objets du domaine et l'accès aux données à la couche de persistance. La couche `Application` utilise directement la couche `Domaine`, mais elle est rendue indépendante de la couche `Persistance` par le biais des interfaces `IRepository`, `IUnitOfWork` et `IDataFactory`, comme montré ci-dessous.



INSTRUCTIONS Votre tache est d'implémenter les classes de la couche applicative contenue dans le dossier `Service`. Dans un premier temps, concentrez vos efforts sur `TraiterEmprunt()` et `TraiterRetour()`. Puis, passez à l'implémentation des autres cas d'utilisation, qui consistent simplement à gérer la création, la modification et la suppression des adhérents, des ouvrages et des exemplaires. Le développement piloté par les tests est toujours conseillé.

4.3.4 Conception de l'interface utilisateur

La couche applicative est une façade du système qui expose les fonctions métier recensées dans le diagramme des cas d'utilisation. Elle est donc le point d'entrée idéal pour les interactions avec l'interface utilisateur. En l'occurrence, celle-ci consiste d'une fenêtre principale qui affiche les informations importantes et qui contient les boutons pour envoyer des commandes.



INSTRUCTIONS Votre tache est de compléter l'implémentation de la classe `Fenetre` dans le dossier `IHM`, afin de traduire les événements émis par l'interface utilisateur en appels aux fonctions exposée par la couche applicative. Vous êtes autoriser à créer des nouvelles fenêtres de dialogue pour gérer la création et la modification des adhérents, des ouvrages et des exemplaires.

ATTENTION : Pour afficher les informations dans l'IHM, il faut redéfinir la méthode `ToString()` dans les classes du domaine.