

Guide du développeur

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Commentaire** | **Auteur** | **Société** |
| 0.1 | 16/06/2017 | Version initiale | Pôle architecture 52 | EID |
| 0.2 | 21/06/2017 | Modifications après présentation IBM | Vincent JACQUOT | EID |
| 0.3 | 23/06/2017 29/06/2017 | Règles d’utilisation des énumérations Règles de choix des identifiants | Vincent JACQUOT | EID |
| 0.4 | 13/07/2017 | Conventions de mapping | Alexandre HUBER | EID |
| 0.5 | 26/07/2017 | Règles d’usage des exceptions Règles d’implémentation d’un client WS Membres publics virtuels pour les entités | Vincent JACQUOT | EID |
| 0.6 | 14/08/2017 | Utilisation des attributs de personnalisation des noms des objets BDD | Florent MILLET | EID |
| 0.7 | 18/08/2017 | Précisions sur le rôle des Repositories et précisions sur les versions .NET et EF. | Vincent JACQUOT | EID |
| 0.8 | 19/09/2017 | Renommage des couches DDD. | Bertrand SCHNEIDER | EID |
| 0.9 | 20/10/2017 | Ajout de la gestion d’évènement. | Bertrand SCHNEIDER | EID |
| 0.9.1 | 07/12/2017 | Correction identité des entités | Tony THONG | EID |
| 0.9.2 | 11/12/2017 | Ajout de la description du one to many | Tony THONG | EID |

[Chapitre 1. Théorie 3](#_Toc500829605)

[1.1 Architecture applicative 3](#_Toc500829606)

[1) Approche DDD : Domain Driven Design 3](#_Toc500829607)

[2) Couches applicatives 6](#_Toc500829608)

[3) Système distribué 7](#_Toc500829609)

[1.2 Process de développement 9](#_Toc500829610)

[1) Test Driven Development 9](#_Toc500829611)

[2) Analyse statique 10](#_Toc500829612)

[3) Intégration continue 11](#_Toc500829613)

[Chapitre 2. Implémentation 12](#_Toc500829614)

[2.1 Fil rouge : les solutions exemple 12](#_Toc500829615)

[2.2 Conventions de codage 13](#_Toc500829616)

[2.3 Implémentation du processus de développement 14](#_Toc500829617)

[1) Tests unitaires 14](#_Toc500829618)

[2) Analyse statique 14](#_Toc500829619)

[3) Intégration continue 16](#_Toc500829620)

[2.4 Implémentation de l’architecture applicative 17](#_Toc500829621)

[1) Organisation des sources 17](#_Toc500829622)

[2) Technologies 20](#_Toc500829623)

[3) Infrastructure 22](#_Toc500829624)

[4) Projet domaine 23](#_Toc500829625)

[5) Projet des interfaces 37](#_Toc500829626)

[6) Projet des services 39](#_Toc500829627)

[7) Projets des systèmes externes 41](#_Toc500829628)

[8) Projet Web ASP.NET 42](#_Toc500829629)

[9) Projets des tests unitaires 44](#_Toc500829630)

[10) Gestion des exceptions 45](#_Toc500829631)

# Théorie

Dans ce chapitre nous allons décrire les concepts théoriques utilisés que nous préconisons chez EIT. L’objectif est de garantir une homogénéité, une haute qualité et une rapidité dans le développement. L’implémentation des concepts, décrite dans la section suivante, pourra varier selon le contexte. Cependant la finalité devra être la même.

Pour cela nous commencerons par décrire notre architecture applicative dans la section 2.1. Nous allons ensuite dans la section 2.2 décrire le process de développement.

## Architecture applicative

Pour concevoir l’application nous pouvons nous appuyer sur l’approche DDD, cela permettra de bien identifier les différents contextes et définir leurs limites.

### Approche DDD : Domain Driven Design

Le DDD (Domain Driven Design) est une approche de conception logicielle. Elle définit un certain nombre de bonnes pratiques et de patterns visant à faciliter et simplifier la création d’applications complexes.

Le sujet est vaste et ne saurait être résumé en quelques lignes. Aussi si vous n’êtes pas familier avec cette approche nous vous invitons à consulter les liens suivant:

<http://blog.infosaurus.fr/public/docs/DDDViteFait.pdf>

<http://cdiese.fr/domain-driven-design-en-5-min/>

L’aspect du DDD qui nous intéresse spécifiquement dans ce document est l’aspect architecture applicative.

Vous trouverez ci-dessous une liste non exhaustive des termes du DDD qu’il est important de connaître pour la suite :

#### Domaine

Par domaine, on sous-entend « domaine métier », c’est-à-dire toute la logique métier autour de laquelle sera développée une application.

#### Entité / Objet valeur

Le DDD divise nos objets métiers en deux catégories : les entités et les objets valeurs.

Les entités et les objets valeur sont des objets au sens POO du terme, ce sont des instances d'une classe. Ils sont tous une représentation d'une partie de notre métier (domaine). La principale différence entre entités et les objets valeur réside dans la notion "d'identité".

Les entités possèdent un identifiant unique, alors que les objets valeurs sont simplement utilisés par des entités pour structurer des données (une date, une adresse postale, etc.).

D’ailleurs deux entités se comparent entre elles via leur identifiant alors que les objets valeurs se comparent via l’intégralité de leur contenu. Les objets valeurs seront intégrés dans les entités, le fait de les distinguer permet une meilleure compréhension et manipulation des données.

**Exemple** :

Un objet personne qui contient des données comme le n° de sécurité sociale, le prénom, le nom, etc. est une entité identifiable grâce au n° de sécurité sociale qui est unique. Deux personnes qui ont exactement le même prénom, nom, etc. peuvent être différentes si leur n° de sécurité sociale est différent, c’est le comportement souhaité.

Un objet localisation qui contient une longitude et une latitude est un objet valeur car l'ensemble des champs forment la valeur de l’objet. Deux localisations avec la même longitude et la même latitude sont égales. Dans ce cas il n’y a pas d’identifiant unique pour les différencier et il n’y a pas d’intérêt d’en ajouter un.

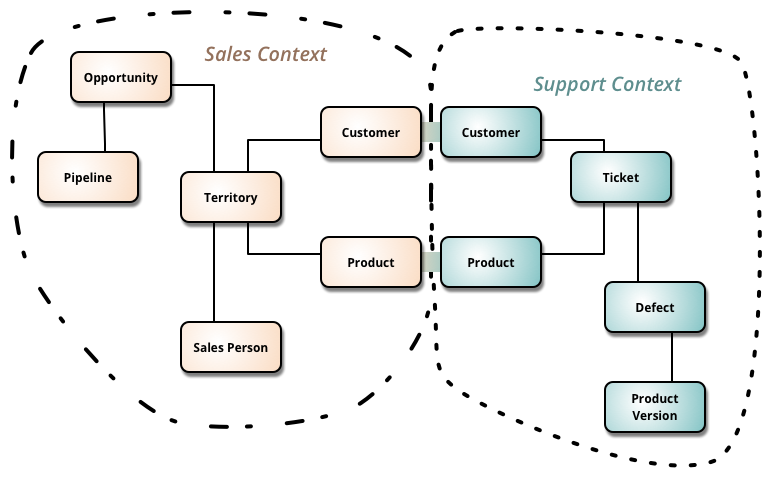
#### Agrégat

Un Agrégat est un groupe d’entités associées qui sont considérées comme un tout unique vis-à-vis des modifications de données. Chaque entité du domaine appartient à un et un seul Agrégat. L’Agrégat est démarqué par une frontière qui sépare les objets situés à l’intérieur de ceux situés à l’extérieur. L’entité racine est le point d’entrée pour accéder aux objets situés à l’intérieur de l’Agrégat.

Chaque Agrégat possède une entité racine sans laquelle les autres entités de l’Agrégat ne peuvent pas exister. Si l’on souhaite apporter des modifications aux objets de l’agrégat, il faut passer par les méthodes de l’objet racine.

#### Bounded Context

Le bounded context ou contexte borné en français est le résultat d’un découpage d’un domaine trop vaste en plusieurs modèles plus petits. Chaque contexte borné a des frontières bien précises avec les autres modèles, et utilise ses propres termes pour nommer ses entités. Il a au final une meilleure cohésion.



Exemple de bounded contexts[[1]](#footnote-1) : Sales Context et Support Context

Les groupes de personnes qui gèrent les ventes (Sales en anglais) n’utilisent pas le même vocabulaire que ceux du support. Il est donc plus difficile de faire un modèle unifié, d’où la nécessité de bien séparer les deux modèles.

Les relations sont explicites, les deux BC (bounded context) partagent le même identifiant pour les entités Customer et Product.

#### Repository

Un Repository est un objet qui contient les méthodes pour rechercher les entités racines, les ajouter et les supprimer de la persistance. Chaque agrégat dispose de son Repository, il s’occupera de persister l’agrégat dans son ensemble. Les responsabilités du Repository sont limitées à la recherche et à la persistance des données, ils ne sont pas chargés de manipuler les entités ou d’interagir avec d’autres agrégats.

#### Service métier

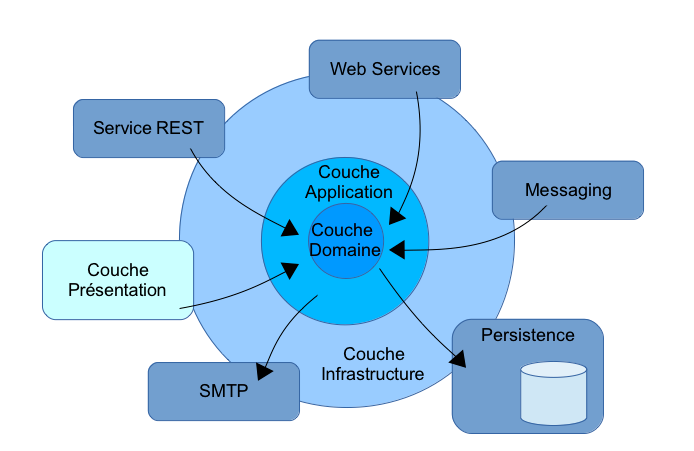
Les services métiers sont des objets qui ont pour responsabilité d’héberger la logique métier qui ne trouve pas sa place au sein des entités ou des Repositories.

#### Service applicatif

Les services applicatifs ont la responsabilité de coordonner les activités de l’application. Les services applicatifs exposent un sous ensemble des fonctionnalités du métier mais ne contiennent pas de logique métier et ne maintiennent aucun état des objets du domaine.

Ils portent également des fonctionnalités techniques transverses telles que la gestion des transactions et la sécurité. Enfin, leur responsabilité est également de récupérer, ajouter ou supprimer les objets du domaine via les Repositories.

#### Architecture en couches



Le DDD propose une architecture applicative en plusieurs couches:

* Domaine
* Infrastructure
* Application
* API

Celles-ci seront décrites plus en détail dans la section suivante.

### Couches applicatives

#### Domaine

La couche domaine a pour objectif d’héberger et d’isoler l’intégralité de la logique métier.

L’isolation se matérialise par la création d’interfaces pour toutes les dépendances externes du métier. Par exemple, lorsqu’un processus métier requiert l’appel d’un système externe, le domaine déclare une interface représentant ce dont le domaine a besoin du système.

Ainsi, le code métier est isolé dans son propre contexte. Il peut utiliser ses propres termes et définitions pour nommer les entités et les processus sans risque de conflit avec ceux des références externes. De plus, ce principe d’isolation facilite grandement la testabilité du code métier.

Il en va de même pour la persistance. Le métier déclare ses besoins en termes de persistance de son point de vue comme pour un système externe via un repository.

L’implémentation de cette interface est fournie par la couche infrastructure selon le contexte (production, test, …).

#### Infrastructure

La couche infrastructure est une couche transverse. Elle a pour objectif de fournir un support aux autres couches en proposant des services techniques indépendants de l’application (persistance BDD, IOC).

Les frameworks du marché utilisent généralement la technique d’injection de dépendance pour matérialiser cette couche infrastructure.

La couche infrastructure accueille également les implémentations des dépendances externes définies par le métier.

Il s’agit :

* De l’implémentation des « Repositories » pour l’accès aux données
* De l’implémentation des systèmes externes

#### Application

La couche application héberge la partie « publique » du métier et l’expose via des services applicatifs.

Ces services applicatifs portent également la couche technique qui est autour d’un service telle que la gestion de la transaction ou la sécurité.

Il est important de rester vigilant sur la non présence de code métier ou de logique d’accès aux données à l’intérieur de cette couche.

#### API

La couche API également appelée couche de présentation est la couche qui expose la couche application vers l’extérieur. Elle permet de choisir les services que l’on souhaite exposer ainsi que la technologie d’exposition selon le client.

### Système distribué

#### Particularités d’un Système distribué

Différents Bounded Contexts peuvent être amenés à devoir se coordonner tout en étant exécutés dans des contextes, des programmes ou sur des machines séparés. Ils forment alors un système distribué composés de nœuds indépendants qui communiquent entre eux.

**Portée de la transaction**

Travailler dans un système distribué nécessite de comprendre la portée limitée des transactions.

Une transaction est un ensemble de tâches liées entre elles qui échouent (abort) ou réussissent (commit) globalement, entre autres choses. Pour mettre en place un système transactionnel, il faut un moyen de synchroniser les différentes tâches de la transaction. Dans un système distribué, maintenir une telle synchronisation s’avère complexe et peu fiable. Il faut donc s’abstenir d’utiliser une transaction commune à travers plusieurs Bounded Contexts.

Si un processus métier effectue des modifications qui concernent plusieurs Bounded Contexts, alors il ne pourra pas compter sur une transaction pour effectuer un retour arrière intégral en cas d’échec. Dans ce cas, le code métier doit pouvoir gérer des échecs partiels ou des annulations partielles d’opérations.

**Gestion des pannes**

Il est faut également tenir compte des pannes et des indisponibilités potentielles des autres nœuds avec lesquels on communique. Par défaut, il faut considérer que le nœud hébergeant un autre Bounded Context peut et va tomber en panne et il faut concevoir son système pour qu’il soit résilient.

Le comportement à adopter en cas d’indisponibilité d’un autre nœud est une problématique métier.

Si la communication avec un autre nœud est critique, alors le métier peut décider de lever directement une erreur. Mais le métier peut également déterminer que la communication avec l’autre nœud n’a pas besoin d’être immédiate ou n’est pas indispensable. Le système pourra alors fonctionner en mode dégradé.

Voici des exemples de fonctionnement dégradé :

* En cas d’échec de la récupération immédiate d’un catalogue, un système pourra s’appuyer sur les dernières données mises en cache dont il dispose.
* En cas de problème sur un système de scoring, un système de commande pourra décider de s’en passer et de continuer à fonctionner sans validation.

**Avantages et inconvénient d’un système distribué**

Avantages :

* Gestion complète des erreurs, amène à ne pas oublier les cas d’erreur
* Fonctionne en mode dégradé
* Fonctionne en mode synchrone et asynchrone
* Performance globale répartie sur plusieurs systèmes, meilleure qu’un système centralisé
* Coût des machines : un gros serveur coûte plus cher que plusieurs serveurs
* Scalabilité : les ressources peuvent être augmentées rapidement

Inconvénients :

* Développement complexe
* Fiabilité technique moindre, le réseau est plus sollicité
* Sécurité à mettre en place sur tous les systèmes

#### Appels synchrones

La communication synchrone entre deux Bounded Contexts est adaptée lorsqu’un retour immédiat d’information est nécessaire. En fonction du niveau de séparation entre les différents Bounded Contexts, différentes solutions sont disponibles.

#### Appels en mode asynchrone

Les appels asynchrones entre les Bounded Contexts sont basés sur un mode d’échange de *messages* ou d’*évènements* en mode *publish-subscribe*[[2]](#footnote-2). Dans un tel mode de fonctionnement, des systèmes appelés *publishers* (éditeurs en français) émettent des messages dans un topic. Des systèmes appelés *subscribers* (abonnés en français) peuvent s’abonner à un topic et consommer les messages émis par les éditeurs.

Les Bounded Contexts peuvent utiliser des échanges en mode message pour communiquer entre eux lorsqu’ils n’ont pas besoin d’un retour immédiat et s’ils veulent bénéficier des avantages d’une communication asynchrone.

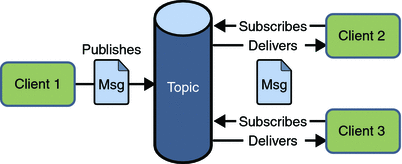


Figure 1 : Schéma du pattern Publish-Subscribe

Avantages :

* Les éditeurs émettent des messages sans se soucier de la disponibilité des abonnés ou du temps de traitement des messages par les abonnés. En particulier, ils peuvent continuer à fonctionner en cas de disfonctionnement d’un abonné.
* Un client n’a pas besoin d’être constamment disponible car il peut dépiler les messages d’un topic de manière différée.

Inconvénients :

* Le métier doit tenir compte de la nature asynchrone des opérations et doit dans certains cas prévoir des opérations d’annulation ou de retour arrière.

## Process de développement

### Test Driven Development

Le TDD (Test Driven Development) est une philosophie de développement logiciel essentiellement basée sur l'écriture préliminaire des tests unitaires pour faciliter la production de code valide et testable. En effet les tests unitaires doivent être produits avant même l’implémentation finale de la fonctionnalité car ils permettent d'orienter positivement la réalisation. Le cycle de développement du TDD nécessite d'avoir spécifié les fonctionnalités et de disposer d’un ensemble de cas d'usage. Il consiste ensuite à :

* Créer un test unitaire qui échoue car la fonctionnalité n’est pas encore implémentée (et qui fonctionne si la fonctionnalité est déjà implémentée)
* Implémenter le code pour réaliser la fonctionnalité de la manière le plus simple possible pour que le test n'échoue plus
* Refondre l'implémentation pour qu'elle soit de meilleure qualité

Lorsque les besoins changent les cas d'usage et nécessitent une évolution du code source, il suffit de répéter le cycle de développement du TDD. Ce cycle souvent nommé Red-Green-Refactor contribue donc à orienter la manière d'implémenter le code.

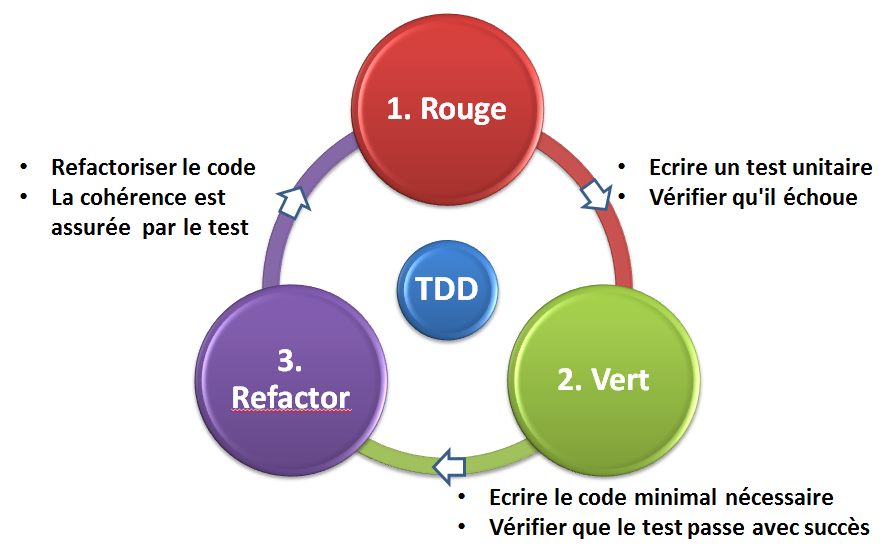


Figure 2 - Schéma du processus Red-Green-Refactor

Les avantages du TDD sont :

* D’avoir naturellement une qualité plus importante du code
* Modularité du code
* Développer le strict minimum en termes de fonctionnalités
* Confiance dans le code
* Rapidité de refonte du code
* Disparition de la peur de modifier le code
* Garantie de non régression

**Liens utiles sur l’approche TDD :**

* Définition : <https://fr.wikipedia.org/wiki/Test_driven_development>
* Introduction : <http://igm.univ-mlv.fr/~dr/XPOSE2009/TDD/pagesHTML/PresentationTDD.html>
* Mise en œuvre du TDD dans le cycle de développement : <http://www.les-traducteurs-agiles.org/tdd/2015/01/06/les-cycles-du-developpement-pilote-par-les-tests.html>
* Exemple concret d’application du TDD : <http://bruno-orsier.developpez.com/tutoriels/TDD/pentaminos/>

### Analyse statique

L’analyse statique est une technique qui consiste à explorer le code source pour calculer des métriques et détecter d’éventuelles erreurs ou problèmes de lisibilité. Cette technique vise à compléter les vérifications effectuées par le compilateur et par les tests unitaires en ajoutant des ensembles de règles supplémentaires pour garantir la qualité du code et le respect des normes de codage.

Contrairement à des techniques comme le débogage ou le profiling, l’analyse statique n’exécute pas le code analysé. A l’instar de la compilation, elle permet de détecter d’éventuelles erreurs au plus tôt.

### Intégration continue

L'intégration continue est un ensemble de pratiques utilisées en développement logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.

L’intégration continue passe par la mise en place d’une brique logicielle qui va permettre d’automatiser les tâches de compilation, de tests unitaires et fonctionnels, de validation du produit etc. Les prérequis de cette pratique sont l’utilisation d’un gestionnaire de source pour versionner le code source, le commit fréquent des développeurs et l’implémentation de tests d’intégrations pour valider l’application.

Les principaux avantages de cette technique de développement sont le test immédiat des modifications et la détection et la résolution des éventuels problèmes d’intégration au plus tôt.

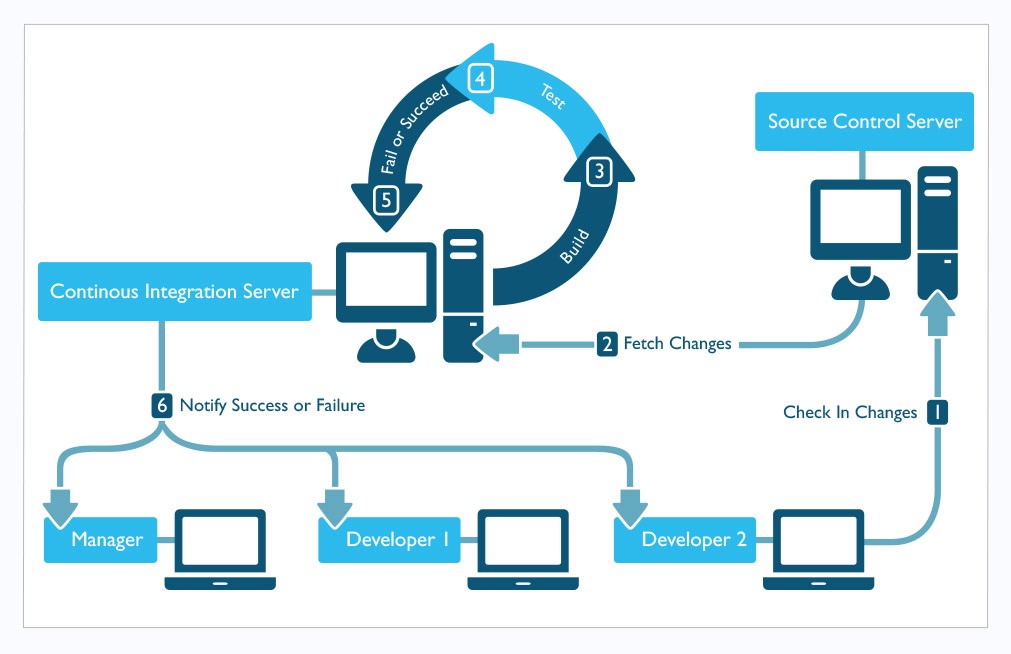


Figure 4: Schéma Intégration Continue

**Liens utiles sur l’intégration continue :**

<http://skalp.developpez.com/traductions/martin-fowler-integration-continue/#LV>

<https://openclassrooms.com/courses/integration-continue>

# Implémentation

## Fil rouge : les solutions exemple

Afin de fournir une base de travail concrète pour expliquer l’implémentation d’un domaine applicatif, deux solutions Visual Studio 2015 d’exemple sont mises à disposition sous la forme d’un dépôt git. Les solutions contiennent un projet web et un projet de tests unitaires qui peuvent être exécutés sur un poste de développement.

Les solutions exemple implémentent des fonctionnalités simplifiées de gestion de commandes, d’articles et de colis. L’ensemble est composé de deux applications : l’une permettant de gérer les commandes, et l’une gérant les articles et les colis.

L’application des commandes permet de gérer des commandes composées d’articles et associées à un tiers. L’application de la logistique gère les articles et les colis. Ces deux applications communiquent entre elles.



Figure 5 - Diagramme de classes de l'application des commandes



Figure 6 - Diagrammes de classes de l'application de la logistique

## Conventions de codage

Indépendamment du processus de développement et des technologies utilisées, le développement applique des conventions de codage. Une partie de ces conventions est vérifiée par l’analyse statique (voir [Analyse statique](#_Analyse_statique)).

En exemple, les conventions suivantes devront être respectées :

* Les noms des classes, membres, variable ou paramètres ne peuvent contenir que des caractères alphanumériques (pas de caractères accentués).
* Les propriétés et les méthodes débutent par une majuscule.
* Les champs et les paramètres débutent par une minuscule.
* Les membres d’instance qui appartiennent à l’objet lui-même sont toujours préfixés par le mot clé this.
* Les membres statiques sont toujours préfixés par le nom du type auquel ils appartiennent.
* Il n’y a jamais plus d’une instruction par ligne.
* Il n’y a jamais plus d’une déclaration par ligne.
* Les types publics et leurs membres publics possèdent des commentaires de documentation XML.
* Les variables d’itération dans les instructions foreach sont typées avec le mot clé var pour éviter les conversions implicites.
* Les types sbyte, ushort, uint, et ulong ne sont pas utilisés.
* Les « method groups » sont utilisés à la place des constructeurs de délégués.
* Le multithreading n’est pas utilisé sauf pour l’asynchronisme avec les méthodes « async » utilisées avec « await ».

## Implémentation du processus de développement

### Tests unitaires

#### Définition d’un bon test unitaire

Les tests unitaires ont pour but de valider le bon fonctionnement d’une partie précise du code. Chaque test se fait sur une méthode, il doit rester limité pour être compréhensible. Un bon test unitaire doit s’exécuter rapidement, être simple, lisible, robuste et maintenable. Le test permet de valider qu’une spécification a bien été réalisée et sécurise la maintenance. Le test ne doit pas prendre en compte les dépendances externes, elles seront simulées par des classes de Mock.

#### Les classes à tester

Toutes les classes publiques devraient être testées et donc référencées dans un projet de test. Chaque projet à tester doit avoir son propre projet de tests de manière à pouvoir conserver la même structure d’espaces de noms et de classes. Il sera alors très facile de retrouver les tests associés à une classe.

Les appels à des classes externes au projet à tester doivent être interfacés. L’interface est là pour permettre de substituer la classe externe par une classe de substitution qui a aura le comportement attendu afin de réaliser les tests uniquement sur la partie de code à tester.

Les interactions entre les classes publiques doivent passer par une interface.

#### Objets mock

Les objets Mock sont là pour substituer les classes qui ne font pas l’objet du test. Il est alors possible dans chaque test unitaire de retourner le résultat attendu pour valider le test.

Les classes factices peuvent être créées de toute pièce en implémentant l’interface présente dans le projet à tester ou être créées automatiquement par un générateur de Mocks à partir d’une interface.

#### Liens

Test unitaire : <https://fr.wikipedia.org/wiki/Test_unitaire>

Mock : <https://fr.wikipedia.org/wiki/Mock_(programmation_orient%C3%A9e_objet)>

Tuto : <https://openclassrooms.com/courses/programmez-en-oriente-objet-avec-c/les-tests-unitaires-5>

### Analyse statique

#### SonarQube

L’analyse statique du code sera effectuée par l’outil SonarQube.

<https://fr.wikipedia.org/wiki/SonarQube>

<https://www.sonarqube.org/>

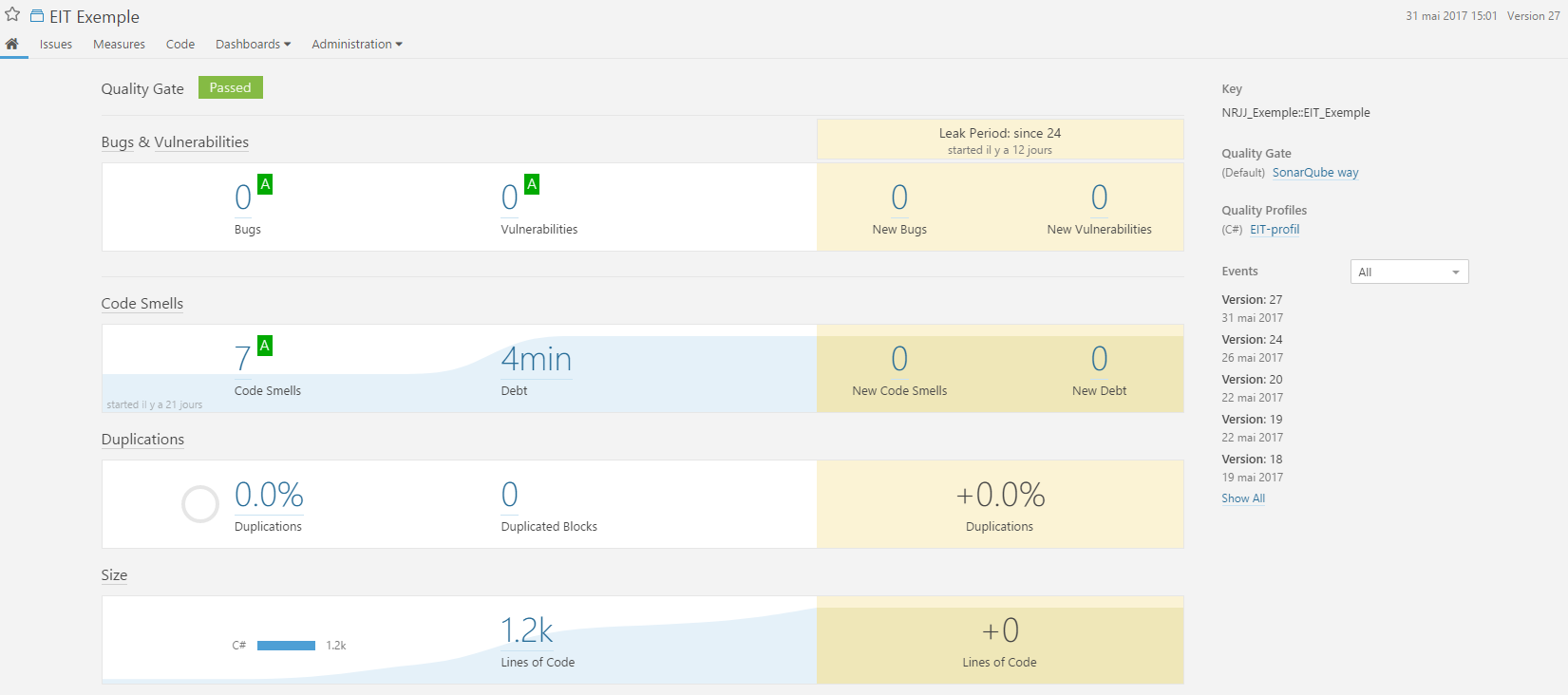
SonarQube est un outil visant à évaluer la qualité du code via la définition et l’évaluation de plusieurs règles de développements.

**Exemple :**

* *La méthode "ToString()" ne doit pas retourner null*
* *La recursion ne doit pas être infinite*

SonarQube sera intégré au process d’intégration continue, ainsi chaque archivage de code donnera lieu à un rapport d’analyse affichant entre autre :

* Les règles Sonar enfreintes classées par catégories (bugs, vulnerabilities, code smells)
* Le résultat des tests unitaires ainsi que le pourcentage de couverture du code
* Le pourcentage de duplication du code



#### Quality Gate

La Quality Gate est un ensemble de conditions évaluées lors de chaque analyse Sonar.

Celles-ci définissent un niveau de qualité que l’application doit respecter lorsque chaque développeur archive du code.

Si toutes les conditions de la Quality Gate sont respectées elle sera marquée comme « Passed » et inversement si une ou plusieurs conditions ne sont pas respectées elle sera marquée comme « Failed ». A terme, le passage de la Quality Gate sera une condition au déploiement d’une application.

#### SonarLint

SonarLint est une extension pour VisualStudio permettant d’avoir un retour directement dans l’IDE de l’évaluation des règles définies dans SonarQube.

<http://www.sonarlint.org/visualstudio/index.html>

#### Lien vers les règles sonar

Les règles de développements sélectionnées pour l’analyse du code ont été documentées à l’emplacement suivant :

<http://xcmci09h.formation.cm-cic.fr:8085/pages/viewpage.action?pageId=2785284>

### Intégration continue

Notre serveur d’intégration continue est [GoCD](https://www.gocd.org/). Il permet de construire des pipelines, c’est-à-dire des enchaînements d’étapes qui peuvent s’exécuter soit manuellement, soit automatiquement lors un commit dans les sources, à heure fixe, ou suite au succès d’un autre pipeline.

Nous pourrions être amenés à utiliser un autre logiciel d’intégration continue, les fonctionnalités resteront les mêmes.

Pour chaque application, trois pipelines sont en place dans GoCD :

* Le pipeline de compilation. Il se déclenche lorsque GoCD détecte qu’un nouveau commit/changeset a été poussé sur la branche de développement. Ce pipeline enchaîne trois tâches : la compilation, l’exécution des tests unitaires et le packaging de l’application. Le résultat des différentes étapes est consultable par les développeurs, notamment les erreurs de compilation et les tests en échec.
* Un pipeline de déploiement sur le serveur d’intégration. Il se déclenche lorsqu’une exécution du pipeline de compilation se termine en succès. Il déploie la dernière version de l’application sur IIS et vérifie qu’elle s’initialise correctement. La dernière version est alors disponible sur le serveur d’intégration (xcmci52d.formation.cm-cic.fr) et permet aux développeurs de tester leur code lorsqu’il est déployé sur IIS.
* Un pipeline d’analyse statique avec SonarQube. Il se déclenche lorsqu’une exécution du pipeline de compilation se termine en succès. Les résultats de l’analyse sont mis à disposition sur le serveur Sonar (voir [Analyse Statique](#_Analyse_statique)).
* Un pipeline de génération de documentation avec Doxygen. Il se déclenche lorsqu’une exécution du pipeline de compilation se termine en succès. La documentation est mise à disposition en format HTML sur un site web.



Figure 7 - Schéma du déclenchement automatique des pipelines sous GoCD

## Implémentation de l’architecture applicative

### Organisation des sources

#### Versionnage des sources

Les pratiques suivantes sont à respectées lors d’utilisation d’un outil de versionnage :

* Créer des branches pour les différents environnements et pour les futures fonctionnalités développées.
* Commit régulier.
* 1 tâche correspond à 1 commit, ne pas mettre plusieurs tâches dans un même commit.
* Tester le code avant de commit.
* Ecrire un message de commit concis et clair (maximum 50 caractères = guideline Git).
* Définit un workflow commun pour tous les développeurs.

**Pour en savoir plus :**

Bonnes pratiques Git :   
<https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices>

#### Structure de la solution

Les deux solutions de EIT.Exemple se décomposent chacune en 6 projets. Les différentes couches définies par l’approche DDD se répartissent dans les projets de la manière suivante :

* La couche Domaine se trouve dans le projet :
  + EIT.[NomDomaine].Domain qui contient les entités, les objets valeur, les services métier et les interfaces des Repositories qui permettent de manipuler les entités
  + EIT.[NomDomaine].Domain.SharedKernel qui contient les value objects de la couche métier, ces objets sont partagés et peuvent être utilisés dans les entités et les évènements.
  + EIT.[NomDomaine].Domain.SharedKernel.Event qui contient les évènement générés par le couche métier, ces évènements sont partagés et peuvent utiliser les value objects du projet ci-dessus.
* La couche infrastructure est répartie dans les projets suivants :
  + EIT.[NomDomaine].Infrastructure.ExternalService.[NomSystemeExterne] pour l’implémentation des services externes
  + EIT.[NomDomaine].Infrastructure.WebApplication pour la configuration et la construction du conteneur d’injection de dépendances
  + EIT.[NomDomaine].Infrastructure.ServiceWindows spécifique à un service Windows
  + EIT.[NomDomaine].Infrastructure.Repostitory.[TypeRepository] où le type de repository peut être File pour un repo fichier, MapperPerso pour un mapper personnalisé, NHibernate pour un mapper NHibernate, …
* La couche Application est répartie dans les projets suivants :
  + EIT.[NomDomaine].Application.Interface qui contient les contrats que doivent respecter les services (interfaces et objets de présentation)
  + EIT.[NomDomaine].Application qui contient l’implémentation des contrats
* La couche API n’a pas de projet dédié car elle est générée automatiquement dans le projet EIT.[NomDomaine].Infrastructure.WebApplication à partir des interfaces du projet EIT.[NomDomaine].Application.Interface. Cependant, on peut développer de nouvelles API qui se nommeront :
  + EIT.[NomDomaine].API.[TypeAPI] qui décrit une API typée (ex : de type fichier « File » ou WCF « WCF »)

Enfin, il existe des projets de test unitaire :

* EIT.[NomDomaine].Domain.Test pour les tests unitaires des Repositories, des entités et des services métier
* EIT.[NomDomaine].Application.Test pour les tests unitaires des services applicatifs

L’arborescence dans la solution doit suivre l’organisation en couches issue du DDD :

* Solution
  + Domaine (ex : Commande, Article ou Colis)
    - API
      * EIT.[NomDomaine].API.[TypeAPI]
    - Application
      * EIT.[NomDomaine].Application
      * EIT.[NomDomaine].Application.Inerface
      * EIT.[NomDomaine].Application.Test
    - Domain
      * EIT.[NomDomaine].Domain
      * EIT.[NomDomaine].Domain.SharedKernel
      * EIT.[NomDomaine].Domain.SharedKernel.Event
      * EIT.[NomDomaine].Domain.Test
  + Infrastructure
    - ExternalService
      * EIT.[NomDomaine].Infrastructure.ExternalService.[NomSystemeExterne]
    - Repository
      * EIT.[NomDomaine].Infrastructure.Repostitory.[TypeRepository]
    - EIT.[NomDomaine].Infrastructure.ServiceWindows
    - EIT.[NomDomaine].Infrastructure.WebApplication

Dans Visual Studio, les projets peuvent être mis dans des répertoires virtuels. Sur le disque dur les projets restent au même niveau dans un répertoire par projet, chaque répertoire se trouvant dans le dossier racine de la solution.

Les différents projets se reposent sur le Framework EIT.



Figure 8 - Position des projets C# dans le schéma des couches DDD



Figure 9 - Schéma des dépendances entre les projets d’une application

Information sur la concordance avec l’ancienne architecture :

|  |  |
| --- | --- |
| Ancienne archi | Nouvelle archi |
| Application web | EIT.[NomDomaine].Infrastructure |
| NrjMobile.Service.[NomDomaine] | EIT.[NomDomaine].Application EIT.[NomDomaine].Application.Interface |
| NrjMobile.[NomDomaine] | EIT.[NomDomaine].Domain |

### Technologies

#### Vision globale de l’implémentation

Les différents modules d’une application s’appuient sur le framework .NET 4.5.2 et les technologies associées. Le schéma suivant replace les technologies présentées dans le schéma des couches DDD.



Figure 10 - Schéma des couches DDD avec les technologies utilisées par le Framework

#### **WCF - Windows Communication Foundation**

WCF fournit un modèle de programmation unifiée pour construire des applications distribuées. Il va permettre de faire communiquer des composants applicatifs situés sur la même machine ou sur des machines différentes.

Le principal avantage de WCF est qu’il va permettre de faire communiquer des applications de technologies différentes sans modifier le code. Cet avantage permet de découpler le code de l’application et son fonctionnement, du moyen de communications utilisé.

Les trois étapes essentielles du déploiement d'un service WCF :

* A pour Address : définit l'adresse du serveur qui expose le service.
* B pour Binding : définit la façon dont le service sera exposé.
* C pour Contract : définit le contrat (= interface) que le service remplit.

Pour pouvoir utiliser WCF dans vos applications, vous devez ajouter la référence à l’assemblage System.ServiceModel à vos projets.

WCF permet d’exposer les services sous différents protocoles tels que REST, SOAP ou des échanges binaires via un canal nommé. Le framework EIT utilise l’exposition SOAP[[3]](#footnote-3).

**Pour en savoir plus :**

<https://msdn.microsoft.com/fr-fr/library/ms731082(v=vs.110).aspx>

<http://vincentlaine.developpez.com/tuto/dotnet/wcf/#LIII-C-2>

#### Entity Framework

Entity Framework est un outil permettant de mapper des classes .NET à des tables d’une base de données relationnelle. Il propose la création d'un schéma conceptuel composé d'entités qui permettent la manipulation d'une source de données, sans écrire une seule ligne de SQL, grâce à LINQ To Entities. Comparé à d'autres solutions de mapping objet-relationnel (ORM), Entity Framework assure l'indépendance du schéma conceptuel (entités ou objets) du schéma logique de la base de données, c'est-à-dire des tables. Ainsi, le code produit et le modèle conceptuel ne sont pas couplés à une base de données spécifique. Le framework EIT utilise Entity Framework en version 6.

Entity Framework autorise différentes approches pour réaliser le mapping entre des entités et une base de données :

* L’approche « Database First » propose de partir d’un schéma de base de données existant pour générer un fichier de mapping .edmx. les classes C# sont ensuite générées à partir du fichier .edmx.
* L’approche « Model First » propose de partir d’un fichier de mapping .edmx écrit à partir d’un éditeur graphique dans Visual Studio. Le fichier .edmx est ensuite utilisé pour générer des classes C# et un schéma de base de données.
* L’approche « Code First » propose de générer le mapping et le schéma de la base de données à partir des classes C#. Dans cette approche, le mapping est décrit par le code et il n’y a pas de fichier .edmx.

Les développements effectués avec le framework EIT utilisent l’approche « Code First », mais sans générer la base de données. Ceci permet d’avoir le contrôle sur le schéma SQL, tout en évitant de devoir maintenir un fichier .edmx.

Au sein même de l’approche « Code First », il existe plusieurs manières de décrire le mapping entre les entités et la base de données dans le code :

* En décorant les entités avec des attributs.
* En décrivant le mapping à l’aide d’une API fluent.
* En se basant sur des conventions de nommage des entités (mapping par convention).

Les développements effectués avec le Framework EIT utilisent idéalement le mapping par convention. En particulier, il existe une bijection entre le nom des tables et des colonnes du schéma de la base de données et le nom des classes et des propriétés du code C#. Par conséquent, il n’y a pas de mapping à maintenir, ni sous forme de fichier .edmx, ni sous forme d’appels à l’API fluent. En contrepartie, une certaine rigueur est nécessaire dans la gestion et les rattrapages du nom des tables et des colonnes. Il est possible de rencontrer des cas non gérables par les conventions, en particulier lorsque la base de données impose une taille limite pour le nom des tables et des colonnes. Il est alors préconisé d’utiliser les attributs pour indiquer le nom des tables et des colonnes.

**Pour en savoir plus :**

<http://pmusso.developpez.com/tutoriels/dotnet/entity-framework/introduction/>

<https://msdn.microsoft.com/fr-fr/library/gg696172(v=vs.103).aspx>

#### Castle DynamicProxy

Le framework EIT utilise Castle DynamicProxy[[4]](#footnote-4) pour proxiser les services applicatifs et leur ajouter des fonctionnalités transverses telles que la gestion de la transaction, les logs automatiques ou la sécurité.

Ce fonctionnement est transparent pour le développeur.

#### Managed Extensibility Framework

MEF (Managed Extensibility Framework) est un outil fourni avec le framework .NET pour gérer les problématiques de composition et d’extension. Le framework EIT utilise MEF 2 (la version portable de MEF) pour gérer l’injection de dépendances.

L’injection de dépendances permet de fournir l’implémentation d’une interface à l’exécution. Ceci permet notamment un découplage entre le code métier et les dépendances techniques. Le métier peut en effet définir des interfaces que les dépendances techniques doivent implémenter et en obtenir les implémentations lors de l’exécution.

La récupération d’une implémentation depuis une interface est effectuée sur un objet appelé *conteneur*. Le conteneur doit être configuré au démarrage de l’application.

Le conteneur basé sur MEF 2 fourni par le Framework EIT est capable d’instancier des implémentations et de renseigner automatiquement les arguments de leur constructeur. Il faut pour cela que les types des arguments du constructeur soient eux-mêmes des interfaces pour lesquelles une implémentation est disponible dans le conteneur.

#### NUnit

NUnit[[5]](#footnote-5) est un framework très répandu qui permet d’écrire des tests unitaires en C#.

Le framework EIT utilise la version 3 de NUnit.

#### Moq

Moq[[6]](#footnote-6) est une librairie qui permet d’écrire des objets de mock pour substituer l’implémentation d’interfaces dans les tests unitaires. Il s’appuie sur Castle DynamicProxy pour générer les mocks.

Le framework EIT utilise la version 4 de Moq.

### Infrastructure

Le framework est constitué de modules, chaque module est défini par un contrat (une ou plusieurs interfaces) et une implémentation. L’implémentation est sélectionnée via une configuration et instanciée via de l’injection de dépendances. Le cœur du framework est donc le système d’injection de dépendances, les modules sont :

* L’accès aux données, qui contient d’autres modules :
  + L’ORM qui est paramétrable au niveau de :
    - La stratégie de mapping
    - La base de données
* Le système de proxys de services
* Le logging
* L’exposition

Une assembly d’utilitaires (EIT.Tools) vient s’additionner au framework, elle contient des fonctions de base accessibles à n’importe quel module. Elle est utilisée par le framework mais est également à disposition des développeurs.

#### Implémentations par défaut fournies

Le tableau ci-après liste les implémentations fournies de base dans le framework pour chaque module

|  |  |
| --- | --- |
| **Module** | **Implémentation(s)** |
| Accès aux données (EIT.DataAccess) | Entity Framework (EIT.DataAccess.EntityFramework) Collections .NET en mémoire pour le test (EIT.DataAccess.InMemory) |
| Stratégie de mapping | Mapping 1 pour 1 automatique des entités (EIT.DataAccess.EntityFramework.AutoMapping) |
| Base de données | Oracle 12c (EIT.DataAccess.EntityFramework.Oracle) |
| Système de proxy (EIT.Layering) | Castle DynamicProxy (EIT.Layering.Castle) |
| Logging (EIT.Logging) | log4net (EIT.Logging.Log4Net)  NRJJ\_JSA (EIT.Logging.NRJJ\_JSA) |
| Exposition (EIT.Exposition) | WCF + IIS (EIT.Exposition.WCF) |
| Kafka (EIT.Messaging.Kafka) | Broker Apache KAFKA |

### Projet domaine

#### Nommage

Le projet Domaine est nommé suivant le modèle EIT.[NomDomaine].Domain

#### Dépendances paquets/projets/bibliothèques

Afin d’héberger les implémentations des Repositories, le projet Domain a besoin de référencer la librairie d’accès aux données du Framework EIT.

* Les paquets NuGet suivants doivent donc être référencés :
  + EIT
  + EIT.DataAccess

#### Contenu

Le projet Domaine contient les éléments suivant :

* Les classes Entités

Les propriétés de ces classes sont mappées sur les colonnes des tables de la base de données via un ensemble de conventions. Les attributs du namespace System.ComponentModel.DataAnnotations[[7]](#footnote-7) seront utilisés dans le cadre de ces conventions.

**Identité des entités**

Toute entité doit hériter de « EIT.Domain.Entity » et posséder un identifiant. On place l’attribut [Key](https://msdn.microsoft.com/fr-fr/library/system.componentmodel.dataannotations.keyattribute(v=vs.110).aspx) sur la propriété de l’entité qui constitue son identifiant. Il s’agit de la seule entorse au mapping par convention. Positionner cet attribut permet à la fois d’indiquer la clé primaire de l’entité et de marquer la classe comme étant une entité persistante.

Toutes les autres propriétés qui disposent d’un getter et d’un setter sont automatiquement mappées à la colonne correspondante dès lors que son type de données est supporté. L’identifiant d’une entité ne doit pas correspondre à une référence externe comme un code barre, un numéro de téléphone ou un code fourni par un partenaire.

Les propriétés mappées doivent être des auto-properties. Elles doivent disposer d’un accesseur public pour pouvoir être utilisées dans les requêtes LINQ. Le setter des propriétés mappées peut par contre être privé ou protégé.

Les entités doivent être publiques et doivent définir un constructeur sans paramètre et sans implémentation qui peut être protégé ou privé.

Pour des raisons de testabilité, une classe entité ne doit jamais être « sealed » et tous ses membres publics doivent être virtuels. Ceci est en effet nécessaire pour créer des mocks de l’entité.

**Evènements des entités**

Le Framework EIT utilise cet héritage pour la gestion d’évènement. Les évènements sont matérialisés par des champs de l’entité qui devront être de type « event Action<EIT.{NomDomaine}.Domain.SharedKernel.Event.{NomEvenement}> ». Lorsque l’évènement est levé le système de communication publiera automatiquement l’évènement au commit du service applicatif.

Exemple :

public partial class Commande : Entity  
{  
 ...  
  
 /// <summary>  
 /// Sur validation de la commande.  
 /// </summary>  
 public event Action<EIT.Commande.Domain.SharedKernel.Event.CommandeValideeEvent> SurValidation;  
}  
  
public partial class Commande  
{  
 ...  
  
 private sealed class EtatCommandeInitial : EtatCommande  
 {  
 }  
  
 public override void Valider()  
 {  
 ...  
  
 // On lève l'évènement qui sera pris en compte par le framework.  
 var commandeValideeEvent = new SharedKernel.Event.CommandeValideeEvent  
 (  
 this.commande.Evenement()  
 );  
 this.commande.SurValidation(commandeValideeEvent);  
 }  
}

**Noms des tables et colonnes**

Les tables servant à persister les entités en base de données doivent respecter les conventions suivantes :

* Le nom de la table respecte le pattern suivant : *T\_Nom de l’entité*
* Les noms des tables et des colonnes doivent être en majuscule

**Persistance des entités héritées**

Dans le cas où une entité hérite d’une autre entité, les propriétés propres à chaque entité sont persistées dans la table de l’entité correspondante.

Exemple :

public abstract class Article : Entity  
{  
 [Key]  
 public virtual string Cle { get; protected set; }  
 public virtual string Libelle { get; protected set; }  
 public virtual decimal Prix { get; protected set; }  
  
 protected Article()  
 {  
 }  
}  
  
public class Sim : Article  
{  
 public virtual string Operateur { get; protected set; }  
  
 private Sim()  
 {  
 }  
  
 public Sim(string libelle, decimal prix, string operateur) : base (libelle, prix)  
 {  
 if (string.IsNullOrWhiteSpace(operateur))  
 {  
 throw new ArgumentException("L'opérateur est requis.", nameof(operateur));  
 }  
  
 this.Operateur = operateur;  
 }  
}

Une entité de type « Sim » sera persistée dans les deux tables suivantes.

|  |
| --- |
| T\_ARTICLE |
| CLE  LIBELLE  PRIX |

|  |
| --- |
| T\_SIM |
| CLE  OPERATEUR |

**Persistance des associations entre entités**

**Many To One**

Exemple :

public partial class Commande : Entity  
{  
 [Key]  
 public virtual string Cle { get; protected set; }  
   
 public virtual DateTime? DateExpedition { get; protected set; }  
 public virtual Destinataire Destinataire { get; protected set; }  
 public virtual Etats Etat { get; protected set; }  
  
 private Commande()  
 {  
 }  
}

L’association avec l’entité « Destinataire » sera persistée dans la table **T\_COMMANDE** dans une colonne respectant le pattern suivant : *Nom de la propriété de l’entité associée*

|  |
| --- |
| T\_COMMANDE |
| CLE  DATEEXPEDITION  DESTINATAIRE  ETAT |

**Les collections**

Les collections d’entités sont généralement mappées en base de données via une association de type **Many To Many** mais peuvent dans certain cas particulier être mappées selon la cardinalité One To Many.

**Les collections en Many To Many**

La création d’une table d’association est nécessaire pour supporter la liaison entre les tables des entités concernées.

Cette table d’association doit respecter le pattern suivant :

Nom de la table (en majuscule) : T\_*Nom de l’entité possédant la collection\_Nom de la propriété collection*

Nom de la première colonne (en majuscule) : *Nom de l’entité possédant la collection\_PARENT*

Nom de la deuxième colonne (en majuscule) : *Nom de l’entité présente dans la collection\_ELEMENT*

Exemple :

public class Colis : Entity  
{  
 [Key]  
 public virtual string Cle { get; protected set; }  
  
 public virtual Adresse AdresseDeLivraison { get; protected set; }  
 public virtual ICollection<Article> Articles { get; protected set; }  
  
 private Colis()  
 {  
 }  
}

Dans la classe exemple ci-dessus, la collection d’entités « Marchandise » mènera à la création de la table d’association suivante :

|  |
| --- |
| T\_COLIS\_ARTICLES |
| COLIS\_PARENT  ARTICLE\_ELEMENT |

**Personnalisation des noms**

Le module EIT.DataAccess.EntityFramework.AutoMapping se charge de mapper automatiquement les objets métiers du domaine en appliquant ces conventions de nommage. Toutefois dans certains cas, il peut être souhaitable de modifier les noms des tables et colonnes de la base.

Pour cela 3 attributs sont mis à disposition dans le namespace EIT.DataAccess.EntityFramework.AutoMapping.Attributes :

* L’attribut [CustomTableName]

Il permet de personnaliser le nom de la table dans laquelle l’entité est persistée, il doit être appliqué sur la déclaration d’une classe d’entité et paramétré avec un nom de table non null et non vide.

/// <summary>  
 /// Colis.  
 /// </summary>  
 [CustomTableName("T\_ENVOI")]  
 public class Colis  
 {  
 ...  
 }

|  |
| --- |
| T\_ENVOI (au lieu de T\_COLIS) |

* L’attribut [CustomColumnName]

Il permet de personnaliser le nom d’une colonne de table. Il doit être appliqué sur une propriété et paramétré avec un nom de colonne non null et non vide.

Si l’attribut est appliqué sur une propriété de type primitif, il permet de spécifier directement le nom de la colonne dans laquelle cette donnée sera persistée.

/// <summary>  
 /// Marque du terminal.  
 /// </summary>  
 [CustomColumnName("FABRICANT")]  
 public virtual string Marque { get; protected set; }

|  |
| --- |
| T\_TERMINAL |
| FABRICANT (au lieu de MARQUE) |

Si l’attribut est appliqué sur une propriété d’une association de type many-to-one, il permet de spécifier le nom de la colonne dans laquelle la clé de l’entité associée sera stockée.

/// <summary>  
 /// Article correspondant au colis.  
 /// </summary>  
 [CustomColumnName("CLEARTICLE")]  
 public virtual Article Article { get; protected set; }

|  |
| --- |
| T\_COLIS |
| CLEARTICLE (au lieu de ARTICLE) |

Pour les propriétés de type complex type, il est possible de personnaliser le nom de la propriété donnant accès au complex type ainsi que chaque propriété du complex type lui-même mais le nom de la colonne sera toujours formée sur le modèle *Nom de la propriété de type complexe\_Nom de la propriété dans le complex type*. Par exemple, un colis contient une adresse caractérisée par un numéro et une voie :

public class Colis : Entity  
 {  
 /// <summary>  
 /// Adresse de livraison du colis.  
 /// </summary>  
 [CustomColumnName("ADR\_LIV")]  
 public virtual Adresse AdresseDeLivraison { get; protected set; }  
 }

[ComplexType]  
 public class Adresse : IEquatable<Adresse>  
 {  
 /// <summary>  
 /// Numéro.  
 /// </summary>  
 public virtual string Numero { get; protected set; }

/// <summary>  
 /// Voie.  
 /// </summary>  
 [CustomColumnName("RUE")]  
 public virtual string Voie { get; protected set; }  
 }

Cette structure sera mappée dans la table du colis de la manière suivante :

|  |  |
| --- | --- |
| T\_COLIS | |
| ADR\_LIV\_NUMERO | ADR\_LIV\_RUE |

Sans l’usage des attributs les tables attendues seraient :

|  |  |
| --- | --- |
| T\_COLIS | |
| ADRESSEDELIVRAISON\_NUMERO | ADRESSEDELIVRAISON\_VOIE |

* L’attribut [CustomAssociationTable]

Il permet de personnaliser la table d’association dans le cas du mapping d’une collection d’objets. Il doit être appliqué sur une propriété de type collection et être paramétré avec un nom de table non null et non vide, un nom de colonne pour le détenteur de l’élément (non null et non vide) et un nom de colonne pour l’élément détenu dans la collection (non null, non vide et distinct du nom de la colonne du détenteur).

[CustomAssociationTable("T\_COLIS\_MARCHANDISES", "COLIS\_PARENT", "MARCHANDISE\_ENFANT")]  
protected virtual ICollection<Marchandise> MarchandisesInterne { get; protected set; }

|  |  |
| --- | --- |
| T\_COLIS\_MARCHANDISES | |
| COLIS\_PARENT | MARCHANDISE\_ENFANT |

Au lieu de :

|  |  |
| --- | --- |
| T\_COLIS\_MARCHANDISESINTERNE | |
| COLIS\_PARENT | MARCHANDISE\_ELEMENT |

**Les collections en One To Many**

Le one to many peut être utilisé pour matérialiser une relation parent/enfant lorsque l’enfant ne peut appartenir qu’à un seul parent à la fois. De plus dans ce cas précis l’enfant ne peut pas exister sans parent. La relation est donc à la charge du parent.

La création d’une table d’association n’est nécessaire pour supporter la relation. En effet, dans ce cas de figure c’est l’enfant que porte la relation en base de données via une colonne prévue à cet effet.

Lors de la définition d’une collection correspondant à la cardinalité one to many, la collection doit être marquée par l’attribut [OneToMany] définit dans « EIT.Domain ».

/// <summary>  
 /// Shop.  
 /// </summary>  
 public class Shop : Entity  
 {  
 /// <summary>  
 /// Name.  
 /// </summary>  
 [Key]  
 public virtual string Name { get; protected set; }

/// <summary>  
 /// Products.  
 /// </summary>  
 [OneToMany]  
 public virtual ICollection<Product> Products { get; protected set; }  
 }

La convention automatique map dans la table enfant (ici T\_PRODUCT) la clé du parent dans la colonne « SHOP\_PRODUCTS ».

Pour personnaliser le nom de la colonne utilisée, il suffit d’utiliser l’attribut « CustomColumnName » sur la collection.

/// <summary>  
 /// Shop.  
 /// </summary>  
 public class Shop : Entity  
 {  
 /// <summary>  
 /// Name.  
 /// </summary>  
 [Key]  
 public virtual string Name { get; protected set; }

/// <summary>  
 /// Products.  
 /// </summary>  
 [OneToMany]  
 [CustomColumnName("SHOP")]  
 public virtual ICollection<Product> Products { get; protected set; }  
 }

Dans certain cas il est utile d’avoir la propriété inverse à la relation, à savoir le many to one définit dans l’enfant qui représente le parent.

/// <summary>  
 /// Shop.  
 /// </summary>  
 public class Product : Entity  
 {  
 /// <summary>  
 /// Name.  
 /// </summary>  
 [Key]  
 public virtual string Name { get; protected set; }

/// <summary>  
 /// Parent shop.  
 /// </summary>  
 public virtual Shop ParentShop { get; protected set; }  
 }

Lorsque cette relation existe il n’est pas nécessaire d’avoir une colonne supplémentaire pour persister cette relation car la même colonne peut être utilisée. Il suffit dans ce cas de préciser le lien qu’il y a entre ces 2 relations à l’aide de l’attribut [InverseProperty] définit dans l’espace de nom « System.ComponentModel.DataAnnotations.Schema » et à placer sur la collection. L’attribut requiert une chaine correspondant à la propriété inverse (le many to one). Pour fiabiliser la déclaration il est vivement recommandé d’utiliser le mot clé nameof au lieu d’écrire une chaine en dur.

/// <summary>  
 /// Shop.  
 /// </summary>  
 public class Shop : Entity  
 {  
 /// <summary>  
 /// Name.  
 /// </summary>  
 [Key]  
 public virtual string Name { get; protected set; }

/// <summary>  
 /// Products.  
 /// </summary>  
 [OneToMany]  
 [InverseProperty(namesof(Product.ParentShop))]  
 public virtual ICollection<Product> Products { get; protected set; }  
 }

Lorsque l’attribut [InverseProperty] est utilisé, c’est le many to one qui porte le mapping. Cela signifie que la colonne utilisé sera « PARENTSHOP ».

Dans le cas où l’on souhaite personnaliser le nom de la colonne, il suffit d’utiliser l’attribut [CustomColumnName] sur le many to one.

/// <summary>  
 /// Shop.  
 /// </summary>  
 public class Product : Entity  
 {  
 /// <summary>  
 /// Name.  
 /// </summary>  
 [Key]  
 public virtual string Name { get; protected set; }

/// <summary>  
 /// Parent shop.  
 /// </summary>  
 [CustomColumnName("SHOP")]  
 public virtual Shop ParentShop { get; protected set; }  
 }

Note : les noms de colonnes ou de tables passés dans les attributs sont **sensibles à la casse**. Un nom avec des minuscules sera traité tel quel par la base de données et il sera donc nécessaire d’entourer les identifieurs de ces objets par des double quotes lors de requêtes manuelles. Pour une meilleure lisibilité, nous recommandons d’utiliser des noms personnalisés uniquement en majuscules.

Note : cette fonctionnalité est essentiellement destinée à traiter les cas où le nom généré par l’AutoMapping est trop long pour le SGBD. L’utilisation des noms générés est recommandée dans toutes les autres situations.

**Persistance des Enums :**

Toute énumération persistée doit définir explicitement les valeurs sous-jacentes de ses constantes. En effet, si les valeurs sous-jacentes ne sont pas explicites, elles peuvent être modifiées si l’ordre des constantes est modifié, ce qui peut rendre les données persistées incohérentes. De plus, il doit toujours exister une constante correspondant à la valeur sous-jacente « 0 », car elle correspond à la valeur par défaut d’une valeur d’énumération non initialisée. L’exemple ci-dessous montre la définition correcte d’une énumération :

/// <summary>  
/// Etats possibles pour une commande.  
/// </summary>  
public enum Etats  
{  
 Initial = 0, // Valeur par défaut de l’énumération, doit toujours être définie  
 Annulee = 1,  
 Validee = 2,  
 Expediee = 3  
}

**Types supportés**

Pour des raisons de maintenance et de compatibilité entre les bases de données, le framework EIT ne gère qu’un un sous-ensemble des types supportés par Entity Framework. Les types gérés sont les suivants :

* Boolean (bool)
* Int16 (short)
* Int32 (int)
* Int64 (long)
* Decimal (decimal)
* String (string)
* DateTime
* Nullable<T> (T?) où T est l’un des types précédents
* Une autre entité
* ICollection<T> où T est une autre entité
* Un type « complexe », c’est-à-dire une autre classe qui n’est pas une entité mais qui contient d’autres propriétés supportées

Tableau de conversion des types Oracle :

|  |  |
| --- | --- |
| .NET | Oracle Data type |
| System.Boolean | NUMBER(1) |
| System.DateTime | TIMESTAMP |
| System.Decimal | NUMBER(x,y) |
| System.Int16 | NUMBER(4) |
| System.Int32 | NUMBER(9) |
| System.Int64 | NUMBER(18) |
| System.String | VARCHAR2(x) |

La norme de mapping des types Oracle est disponible à cet emplacement : <https://docs.oracle.com/database/121/ODPNT/entity_framework.htm#ODPNT8273>

Pour plus d’informations vous pouvez consulter la documentation d’Entity Framework sur le MSDN[[8]](#footnote-8).

Le fragment de code ci-dessous est un exemple d’entité :

public class Commande : Entity  
{  
 // Identifiant de l'entité  
 [Key]  
 public virtual string Cle { get; protected set; }  
   
 // Many-To-One  
 public virtual Tiers Destinataire { get; protected set; }  
  
 public virtual Etats Etat { get; protected set; }  
  
 // One-To-Many interne pris en compte par l'ORM  
 internal virtual ICollection<Article> Articles { get; protected set; }  
  
 public Commande(string cle, Tiers destinataire)  
 {  
 if (string.IsNullOrWhiteSpace(cle))  
 {  
 throw new ArgumentException("La clé de la commande est requise.", nameof(cle));  
 }  
  
 if (destinataire == null)  
 {  
 throw new ArgumentNullException(nameof(destinataire), "Le destinataire est requis.");  
 }  
  
 this.Cle = cle;  
 this.Destinataire = destinataire;  
 this.Etat = Etats.Initial;  
 this.Articles = new List<Article>();  
 }  
  
 // Constructeur sans paramètres utilisé par l'ORM  
 private Commande()  
 {  
 }  
  
 // One-To-Many public, pour éviter d'exposer un ICollection  
 IEnumerable<Article> ArticlesCommandes  
 {  
 get { return this.Articles; }  
 }  
}

* Les objets valeurs

Les objets valeurs ne sont pas représentés par des structures mais par des classes à cause des besoins des ORM (Object-Relational Mappings). Cependant, ces classes doivent se comporter comme des types valeurs, d’où l’obligation de surcharger les méthodes Equals() et GetHashCode() définies sur la classe System.Object. Il faut également s’assurer que les objets valeur soient immutables, c’est-à-dire qu’il ne doit pas être possible de modifier leurs champs après leur construction.

**Persistance des objets valeurs**

Les champs des objets valeurs sont mappés sur les colonnes de la table de l’entité parente.

Exemple :

public class Colis : Entity  
{  
 [Key]  
 public virtual string Cle { get; protected set; }  
  
 public virtual Adresse AdresseDeLivraison { get; protected set; }  
 public virtual ICollection<Article> Articles { get; protected set; }  
  
 private Colis()  
 {  
 }  
}  
  
public class Adresse : IEquatable<Adresse>  
{  
 public virtual string Nom { get; protected set; }  
 public virtual string Prenom { get; protected set; }  
 public virtual string Numero { get; protected set; }  
 public virtual string Voie { get; protected set; }  
 public virtual string CodePostal { get; protected set; }  
 public virtual string Ville { get; protected set; }  
  
 private Adresse()  
 {  
 }  
}

Les propriétés de l’objet valeur « Adresse » seront mappées sur les colonnes de la table **T\_COLIS** respectant le pattern suivant : *Nom de la propriété objet valeur\_ Nom de la propriété de l’objet valeur*

|  |
| --- |
| T\_COLIS |
| CLE  ADRESSEDELIVRAISON\_NOM  ADRESSEDELIVRAISON\_PRENOM  ADRESSEDELIVRAISON\_NUMERO  ADRESSEDELIVRAISON\_VOIE  ADRESSEDELIVRAISON\_CODEPOSTAL  ADRESSEDELIVRAISON\_VILLE |

* Les services métier

Sont regroupées sous l’appellation service métier, l’ensemble des méthodes et classes qui comportent du métier mais qui ne sont pas directement rattachées à une entité ou un objet valeur. Il peut s’agir par exemple d’une méthode statique permettant de créer une Entité ou une factory si la création requiert un pattern plus complexe.

* Les interfaces des Repositories

Celles-ci sont utilisées par le code métier et le projet des services applicatifs pour l’accès aux repositories. Les méthodes déclarées dans ces interfaces ne peuvent prendre en arguments que des objets métiers ou des types primitifs. Les méthodes de recherche d’un Repository renvoient un ou plusieurs objets métier. Les interfaces des repositories sont nommées suivant le modèle I[NomAggregat]Repository.

Le fragment de code ci-dessous est un exemple d’interface de Repository :

namespace EIT.Exemple.Domaine.Donnees  
{  
 public interface ICommandeRepository  
 {  
 void AjouterCommande(Commande nouvelleCommande);  
 Commande Obtenir(string cleCommande);  
 }  
}

* L’implémentation des Repositories

Ces classes implémentent les méthodes définies par les interfaces des Repositories. Leur constructeur doit obligatoirement prendre en paramètre un objet de type IDataSource. L’objet IDataSource permet de de faire des recherches à l’aide de requêtes Linq To Entities[[9]](#footnote-9) et d’ajouter et supprimer des entités de la persistance. Les repositories sont nommés suivant le modèle I[NomAggregat]Repository, c’est-à-dire comme leur interface mais sans le préfixe « I ».

Le fragment de code ci-dessous est un exemple d’implémentation de Repository :

using System.Linq;  
using EIT.DataAccess;  
  
namespace EIT.Exemple.Domaine.Donnees  
{  
 public sealed class CommandeRepository : ICommandeRepository  
 {  
 private readonly IDataSource dataSource;  
  
 public RegistreTiers(IDataSource dataSource))  
 {  
 this.dataSource = dataSource;  
 }  
   
 public void AjouterCommande(Commande nouvelleCommande)  
 {  
 if (commande == null)  
 {  
 throw new ArgumentNullException(nameof(nouvelleCommande), "La commande est requise.");  
 }  
  
 this.dataSource.Add(nouvelleCommande);   
 }  
  
 public Commande Obtenir(string cleCommande)  
 {  
 if (string.IsNullOrWhiteSpace(cleCommande))  
 {  
 throw new ArgumentException(nameof(cleCommande), "La clé de la commande est requise.");  
 }  
  
 return this.dataSource.Query<Commande>().Single(commande => commande.Cle == cleCommande);  
 }  
 }  
}

* Les interfaces des systèmes externes

Les interfaces des systèmes externes décrivent les besoins du domaine vis-à-vis des systèmes externes qu’il utilise. Elles ne représentent donc pas l’API du système externe mais une version simplifiée et abstraite de celle-ci spécifiquement adaptée au domaine. Cette étape d’abstraction permet d’éviter d’importer la complexité des dépendances externes dans le domaine et permet également de simplifier l’écriture de mocks.

#### Exemple de code complet

Exemple d’Entité :

public class Commande : Entity  
{  
 // Identifiant de l'entité  
 [Key]  
 public string Cle { get; protected set; }  
   
 // Many-To-One  
 public Tiers Destinataire { get; protected set; }  
  
 public Etats Etat { get; protected set; }  
  
 // One-To-Many interne pris en compte par l'ORM  
 internal ICollection<Article> Articles { get; protected set; }  
  
 public Commande(string cle, Tiers destinataire)  
 {  
 if (string.IsNullOrWhiteSpace(cle))  
 {  
 throw new ArgumentException("La clé de la commande est requise.", nameof(cle));  
 }  
  
 if (destinataire == null)  
 {  
 throw new ArgumentNullException(nameof(destinataire), "Le destinataire est requis.");  
 }  
  
 this.Cle = cle;  
 this.Destinataire = destinataire;  
 this.Etat = Etats.Initial;  
 this.Articles = new List<Article>();  
 }  
  
 // Constructeur sans paramètres utilisé par l'ORM  
 private Commande()  
 {  
 }  
  
 // One-To-Many public, pour éviter d'exposer un ICollection  
 IEnumerable<Article> ArticlesCommandes  
 {  
 get { return this.Articles; }  
 }  
}

Exemple d’interface de Repository :

namespace EIT.Exemple.Domaine.Donnees  
{  
 public interface ICommandeRepository  
 {  
 void AjouterCommande(Commande nouvelleCommande);  
 Commande Obtenir(string cleCommande);  
 }  
}

Exemple de Repository :

using System.Linq;  
using EIT.DataAccess;  
  
namespace EIT.Exemple.Domaine.Donnees  
{  
 public sealed class CommandeRepository : ICommandeRepository  
 {  
 private readonly IDataSource dataSource;  
  
 public RegistreTiers(IDataSource dataSource))  
 {  
 this.dataSource = dataSource;  
 }  
   
 public void AjouterCommande(Commande nouvelleCommande)  
 {  
 if (commande == null)  
 {  
 throw new ArgumentNullException(nameof(nouvelleCommande), "La commande est requise.");  
 }  
  
 this.dataSource.Add(nouvelleCommande);   
 }  
  
 public Commande Obtenir(string cleCommande)  
 {  
 if (string.IsNullOrWhiteSpace(cleCommande))  
 {  
 throw new ArgumentException(nameof(cleCommande), "La clé de la commande est requise.");  
 }  
  
 return this.dataSource.Query<Commande>().Single(commande => commande.Cle == cleCommande);  
 }  
 }  
}

### Projet des interfaces

#### Nommage

Le projet des interfaces est nommé suivant le modèle EIT.[NomDomaine].Application.Interface car c’est un projet qui sera partagé avec les autres domaines et qui doit être séparé de l’implémentation des services applicatifs. Son espace de nom doit être tronqué pour ne pas faire apparaître la notion « Interface » et se positionner au même niveau que l’implémentation des services applicatifs.

#### Dépendances paquets/projets/bibliothèques

Le projet doit des interfaces doit référencer les Assemblies de l’espace de nom System.ServiceModel qui contiennent les attributs qui permettent de décrire les interfaces WCF.

* Assemblies du framework .NET à référencer :
  + System.ServiceModel
  + System.ServiceModel.Web

#### Contenu

Le projet des interfaces contient les contrats des services exposés par l’application. Il ne doit contenir que les objets de présentation (analogues à des DTO) et les interfaces des services applicatifs. Aucun des types définis dans ce projet ne doit posséder de méthodes, il ne s’agit que de la description des contrats.

Les interfaces de services sont nommées suivant le modèle I[NomAggregat]Service. Ils doivent être décorés à l’aide des attributs du [namespace System.ServiceModel](https://msdn.microsoft.com/fr-fr/library/system.servicemodel(v=vs.110).aspx) :

* Chaque service est étiqueté avec l’attribut ServiceContract
* Chaque méthode de service est étiquetée avec l’attribut OperationContract

Les méthodes de services ne peuvent retourner ou prendre en arguments que des objets de présentation définis dans l’assembly elle-même ou des types parmi cette liste :

* System.Byte (byte)
* System.Boolean (bool)
* System.Int16 (short)
* System.Int32 (int)
* System.Int64 (long)
* System.Decimal (decimal)
* System.Guid
* System.DateTime
* Les types d’énumération définis uniquement dans l’assembly elle-même
* Les variantes Nullable des types précédemment cités
* System.String (string)
* Les tableaux des types précédemment cités

Les objets de présentation sont étiquetés avec l’attribut DataContract. Un objet de présentation ne déclare que des propriétés implémentées automatiquement (auto-properties) publiques en lecture-écriture. Les types autorisés pour ces auto-properties sont ceux de la liste précédente. Chaque membre est étiqueté avec l’attribut DataMember.

Tout objet de présentation doit posséder un constructeur public sans paramètre. Lorsqu’aucun constructeur n’est défini explicitement, le framework .NET génère automatiquement ce constructeur.

#### Exemples de code

Exemple d’interface de service applicatif :

using System.ServiceModel;  
  
namespace EIT.Commande.Application  
{  
 [ServiceContract]  
 public interface ICommandeService  
 {  
 [OperationContract]  
 Commande RechercherCommande(string cleCommande);  
  
 [OperationContract]  
 Commande CreerCommande();  
  
 [OperationContract]  
 void DefinirDestinataire(string cleCommande, string cleTiersDestinataire);  
  
 [OperationContract]  
 void AjouterLigneCommande(string cleCommande, Ligne ligne);  
  
 [OperationContract]  
 void SupprimerCommande(string cleCommande);  
 }  
}

Exemple d’objet de présentation :

using System.Runtime.Serialization;  
  
namespace EIT.Commande.Application  
{  
 [DataContract]  
 public sealed class Commande  
 {  
 [DataMember]  
 public string CleCommande { get; set; }  
  
 [DataMember]  
 public string CleTiers { get; set; }  
  
 [DataMember]  
 public EtatCommande EtatCommande { get; set; }  
  
 [DataMember]  
 public string[] ListeCleLigne { get; set; }  
 }  
}

### Projet des services

Le projet des services correspond à la couche services applicatifs de l’approche DDD. Il implémente les contrats des services définis dans le projet des interfaces. Il s’agit d’un projet technique qui ne doit pas contenir de code métier.

#### Nommage

Le projet des interfaces est nommé suivant le modèle EIT.[NomDomaine].Application.

#### Dépendances

Le projet des services référence le projet des interfaces car il doit en implémenter les contrats. Il référence également le projet métier car il doit avoir accès aux interfaces des Repositories, des services externes.

Les dépendances du projet WebApplication de la solution exemple sont donc les suivantes :

* Projets du domaine applicatif :
  + Projet des interfaces EIT.[NomDomaine].Application.Interface
  + Projet métier EIT.[NomDomaine].Domain
  + Projet des value objects EIT.[NomDomaine].Domain.SharedKernel
  + Projet des évènements EIT.[NomDomaine].Domain.SharedKernel.Event
* Assemblies du framework .NET :
  + System
  + System.Core

#### Contenu

Le projet des services contient les classes qui implémentent les contrats définis dans le projet des interfaces. Ces classes contiennent du code technique qui est chargé de récupérer des entités à partir de registres, d’appeler des méthodes sur ces entités ou sur des services métiers, et de faire la conversion entre les objets métier et les objets de présentation. Les services sont nommés suivant le modèle [Nom]Service, c’est-à-dire comme l’interface qu’ils implémentent mais sans le préfixe « I ».

Les services doivent demander les interfaces des repositories et des services externes dont ils ont besoin en argument de leur constructeur. Les implémentations des repositories et des services externes leur seront fournis par injection de dépendances.

Pour la gestion d’évènement il est préférable de créer un nouveau service dédié uniquement aux évènements du domaine qu’il écoute, par exemple ColisPourCommandeService pour le service de gestion des colis qui se déclenche sur des évènements du domaine Commande. Les méthodes de ce service ne doivent prendre qu’un seul paramètre qui correspond à l’objet évènement écouté. Le service n’a pas besoin d’être exposé, il prend en paramètre de construction le service applicatif sans gestion d’évènement qu’il pourra appeler directement.

Les classes qui implémentent les services applicatifs sont proxisées par le framework lors de l’exécution. Le proxy ajoute des fonctionnalités transverses à toutes les méthodes que le service implémente et qui font partie de son contrat. Pour une utilisation standard du framework, les fonctionnalités ajoutées par le proxy sont les suivantes :

* La transaction pour l’accès à la base de données est automatiquement ouverte au début de l’appel d’une méthode d’un service et fermée lorsque la méthode se termine. Si l’appel de se termine sans exception, les modifications sont validées (commit). Si l’appel lève une exception, les modifications sont annulées (rollback). Attention : la transaction n’étant pas distribuée, les modifications effectuées par des appels de systèmes externes ne sont pas annulées.
* Les appels aux méthodes des services sont automatiquement journalisés par le module de log.
* Les appels aux méthodes des services peuvent faire l’objet d’un contrôle d’accès transverse apporté par le module de sécurité.

#### Exemple de code

L’exemple de code suivant montre comment implémenter un service de gestion des tiers dans le projet exemple :

public sealed class TiersService :  
 IGestionnaireTiers // Le service implémente le contrat défini dans le projet EIT.Commande.Application.Interface.  
{  
 // Les registres et systèmes externes sont référencés par des champs privés.  
 private readonly IRegistreTiers m\_RegistreTiers;  
  
 // Le constructeur doit demander l'implémentation des registres et systèmes externes en paramètre.  
 public GestionnaireTiers(IRegistreTiers registreTiers)  
 {  
 if (registreTiers == null)  
 {  
 throw new ArgumentNullException(nameof(registreTiers));  
 }  
  
 this.m\_RegistreTiers = registreTiers;  
 }  
  
 public void CreerTiers(string nom, string prenom)  
 {  
 this.m\_RegistreTiers.CreerNouveauTiers(nom, prenom);  
 }  
  
 public Tiers RechercherTiers(string cle)  
 {  
 Domaine.Tiers tiers = this.m\_RegistreTiers.ObtenirParCle(cle);  
  
 // Le service fait la conversion entre les objets métier et les objets de présentation.  
 return new Tiers  
 {  
 CleTiers = tiers.Cle,  
 Nom = tiers.Nom,  
 Prenom = tiers.Prenom  
 };  
 }  
}

Pour la gestion d’évènement, un exemple de service dédié aux évènements de la commande :

public sealed class ColisPourCommandeService :  
 IColisPourCommandeService // Le service implémente le contrat défini dans le projet EIT.Logistique.Application.Interface.  
{  
 // Le services est référencé par un champ privé.  
 private readonly IColisService colisService;  
  
 // Le constructeur doit demander l'implémentation du service passé en paramètre.  
 public ColisPourCommandeService(IColisService colisService)  
 {  
 if (colisService == null)  
 {  
 throw new ArgumentNullException(nameof(colisService));  
 }  
  
 this.colisService = colisService;  
 }  
  
 public void CreerColisSurCommandeValidee(CommandeValideeEvent commandeValideeEvent)  
 {  
 // Le service fait la conversion entre les objets métier et les objets de présentation.  
 this.ColisService.CreerColis(  
 commandeValideeEvent.Commande.CleCommande,  
 commandeValideeEvent.Commande.AdresseDeLivraison.Presentation(),  
 commandeValideeEvent.Commande.Articles.Select(article => article.CleArticleLogistique).ToArray()  
 ) ;  
 }  
}

### Projets des systèmes externes

Les classes qui permettent de communiquer avec un système externes sont placées dans un projet dédié qui sert de couche d’anti-corruption[[10]](#footnote-10) entre le domaine et l’API du système externe.

#### Nommage

Le projet web est nommé suivant le modèle EIT.[NomDomaine].Infrastructure.ExternalService.[NomSystemeExterne] .

#### Dépendances

Un projet qui permet de communiquer avec un système externe référence le projet métier qui utilise ce système externe. En effet, c’est le code métier qui définit l’interface dont il a besoin pour communiquer avec le système externe. L’implémentation lui sera fournie par l’injection de dépendances.

Les dépendances d’un hypothétique projet de système externe dans la solution exemple sont donc les suivantes :

* Projets du domaine applicatif :
  + Projet métier EIT.[NomDomaine].Domain
* Assemblies du framework .NET :
  + System
  + System.Core
  + Assemblies nécessaires à la communication avec le système externe, par exemple :
    - System.ServiceModel
    - System.Xml
    - …

#### Contenu

Les classes essentielles d’un projet de système externe sont celles qui implémentent les interfaces définies dans le projet métier. Le projet peut également contenir des classes utilitaires qui servent à la communication avec le système externe ou à la conversion des DTO.

L’implémentation de la communication avec le système externe dépend de la technologie utilisée par son API (SOAP, REST, TSI…).

Dans le cas particulier où le système externe est un autre domaine développé avec le framework EIT, un paquet NuGet contenant des contrats de services compatibles avec WCF est mis à disposition. Il est alors préférable d’utiliser les contrats .NET pour générer des proxies à partir des interfaces plutôt que de se baser sur le fichier WSDL.[[11]](#footnote-11)

Le code qui permet de générer un client à partir d’une interface doit être factorisé dans une brique technique afin de centraliser les modifications en cas de changement de mode de communication entre les services. Par exemple, il peut s’agir d’une fabrique ou d’une classe générique qui s’appuie sur l’un des types WCF ChannelFactory ou ClientBase.[[12]](#footnote-12)

### Projet Web ASP.NET

Le point d’entrée de l’application est un projet web ASP.NET. Il contient le code de configuration et d’initialisation qui est exécuté au démarrage de l’application.

**Attention !** Ce projet ne correspond pas à la couche application de l’architecture DDD, il s’agit d’une *application* au sens IIS / ASP.NET qui fait partie de la couche infrastructure.

#### Nommage

Le projet web est nommé suivant le modèle EIT.[NomDomaine].Infrastructure.WebApplication.

#### Dépendances

Comme il s’agit du point d’entrée pour la configuration et l’initialisation de l’application, le projet web références tous les paquets techniques utilisés dans l’application ainsi que tous les projets du domaine applicatif.

Les dépendances du projet web de la solution exemple sont donc les suivantes :

* Projets du domaine applicatif :
  + Projet des interfaces EIT.[NomDomaine].Application.Interface
  + Projet des services EIT.[NomDomaine].Application
  + Projet métier EIT.[NomDomaine].Domain
  + Projet métier des value objects EIT.[NomDomaine].Domain.SharedKernel
  + Projet métier des évènements EIT.[NomDomaine].Domain.SharedKernel.Event
  + Projets des systèmes externes EIT.[NomDomaine].Infrastructure.ExternalService.[NomSystemeExterne]
* Assemblies du framework EIT (via NuGet) :
  + EIT
  + EIT.DataAccess
  + EIT.DataAccess.EntityFramework
  + EIT.DataAccess.EntityFramework.AutoMapping
  + EIT.DataAccess.Oracle
  + EIT.Exposition
  + EIT.Exposition.WCF
  + EIT.Layering
  + EIT.Layering.Castle
  + EIT.Logging
  + EIT.Logging.NRJJ\_JSA
  + EIT.Messaging
  + EIT.Messaging.Description
  + EIT.Messaging.Kafka
  + EIT.Messaging.Transactional
* Assemblies du framework .NET :
  + System
  + System.Core
  + System.Web

#### Contenu

Le projet web contient du code de configuration et d’initialisation qui est appelé depuis la méthode la méthode Application\_Start de classe Global (dans le fichier Global.asax).

La configuration de l’injection de dépendances est un élément central de la phase d’initialisation. Elle se charge d’enregistrer les modules techniques qui composent l’application ainsi que les implémentations des services, des registres et des systèmes externes. Le framework EIT fournit un conteneur d’injection de dépendances basé sur MEF 2 (la version portable de Microsoft Extensibility Framework).

La dernière étape de l’initialisation est la configuration de l’exposition des services applicatifs. Le framework EIT met à disposition un module qui génère automatiquement l’API à partir des services applicatifs enregistrés. Toutes les opérations des services enregistrés sont alors exposées en SOAP avec WCF.

#### Exemple de code

Le fragment de code suivant montre comment configurer le conteneur d’injection de dépendances :

// 1. Déclaration des repositories  
containerBuilder.ExportShared<ITiersRepository, TiersRepository>();  
containerBuilder.ExportShared<ICommandeRepository, RepositoryCommande>();  
  
// 2. Déclaration des systèmes externes  
containerBuilder.ExportShared<ILogistiqueArticleService, LogistiqueArticleService>();  
  
// 3. Déclaration des services  
containerBuilder.ExportShared<ICommandeService, CommandeService>();  
containerBuilder.ExportShared<ITiersService, TiersService>();

Le fragment de code suivant montre comment configurer l’exposition des services applicatifs :

// 1. Enregistrer les services applicatifs et interfaces  
publisher.RegisterService<ICommandeService>();  
publisher.RegisterService<ITiersService>();

### Projets des tests unitaires

Les tests unitaires doivent être placés dans des projets identifiables qui pourront être renseignés dans les outils pour exécuter automatiquement les tests. Ils doivent se trouver dans la même solution que les projets à tester afin que les tests puissent être exécutés à la fin de la compilation.

#### Nommage

Les projets des tests unitaires sont nommés suivant le modèle <NomDuProjet>.Test, par exemple EIT.[NomDomaine].Domain.Test ou EIT.[NomDomaine].Application.Test.

#### Dépendances

Un projet de tests unitaires doit référencer le projet qu’il permet de tester.

Il doit également référencer les paquets NuGet de NUnit 3[[13]](#footnote-13) et de Moq[[14]](#footnote-14), qui sont les outils qui permettent d’écrire respectivement les tests unitaires et les objets mock.

Les dépendances d’un projet de tests unitaires sont donc les suivantes :

* Projets à tester :
  + Par exemple EIT.[NomDomaine].Domain ou EIT.[NomDomaine].Application
* Les paquets NuGet suivants doivent être référencés :
  + NUnit
  + NUnit3TestAdapter s’il n’est pas possible d’installer l’extension en mode administrateur
  + Moq
* Les Assemblies du framework .NET suivantes doivent être référencées :
  + System
  + System.Core
* Il faut également référencer les autres assemblies nécessaires à la compilation

#### Contenu

Le projet des tests unitaires regroupe les classes qui contiennent les tests écrits selon l’approche TDD (voir [Test Driven Development](#_Test_Driven_Development)). Merci de vous reporter à la [documentation NUnit](https://github.com/nunit/docs/wiki/NUnit-Documentation) pour l’utilisation des attributs et des méthodes de test et à la [documentation Moq](https://github.com/Moq/moq4/wiki) pour l’écriture d’objets mock.

Pour chaque classe à tester correspond une classe de test nommée suivant le modèle [ClasseTestee]Test. Les méthodes de test sont nommées de manière explicite suivant le modèle [ActionTestee]\_[ContexteOuArgumentsDuTest]\_[ResultatAttendu].

#### Exemple de code

Le fragment de code suivant montre comment écrire un test avec l’outillage de NUnit :

[Test] // Attribut NUnit pour identifier une méthode test  
public void ValiderCommande\_Simple\_EstEtatValidee()  
{  
 // Préparation du test (Arrange)  
 Tiers tiers = new Tiers("nom", "prenom");  
 Commande instance = new Commande();  
 instance.Destinataire = tiers;  
  
 // Action à valider (Act)  
 instance.Valider();  
  
 // Validation du test (Assert)  
 Assert.That(instance.Etat, Is.EqualTo(Commande.Etats.Validee));  
}

Le fragment de code suivant montre comment vérifier qu’un test lève une exception :

[Test] // Attribut NUnit pour identifier une méthode test

public void ExpedierCommande\_Simple\_LeveInvalidOperationException()

{

// Préparation du test (Arrange)

Commande instance = new Commande();

// Action à valider (Act)

TestDelegate action = () => instance.Expedier();

// Validation du test (Assert)

Assert.That(action, Throws.InvalidOperationException);

}

Le fragment de code suivant montre comment créer et utiliser un objet mock :

[Test]  
public void RechercherTiers\_RetourneTiers()  
{  
 // On construit un objet tiers  
 string cleTiers = Guid.NewGuid().ToString("N");  
 string nomTiers = "Nom";  
  
 Domain.Tiers tiersDomaine = new Domain.Tiers(cleTiers, nomTiers, "Prénom");  
  
 // Fabrication d'un objet mock pour le registre des tiers  
 Mock<IRegistreTiers> registreTiersMock = new Mock<IRegistreTiers>();  
  
 // On définit le retour de la méthode mockée pour une valeur spécifique du paramètre cleTiers  
 registreTiersMock.Setup(registre => registre.ObtenirParCle(cleTiers)).Returns(tiersDomaine);  
  
 // On construit un gestionnaire de tiers en lui passant le registre mocké en paramètre  
 IGestionnaireTiers gestionnaireTiers = new GestionnaireTiers(registreTiersMock.Object);  
  
 Tiers tiersService = gestionnaireTiers.RechercherTiers(cleTiers);  
  
 Assert.AreEqual(tiersService.Nom, nomTiers);  
}

### Gestion des exceptions

La gestion des exceptions doit respecter quelques règles afin de faciliter le déboguage, la journalisation et le traitement des erreurs par les clients. En complément des bonnes pratiques de la documentation standard[[15]](#footnote-15), le développement avec le framework EIT insiste sur les points suivants.

#### Typage des exceptions

Les exceptions qui ont une signification métier doivent être typées afin d’être identifiables lorsqu’elles remontent jusqu’au client. En particulier, les IHM qui souhaitent afficher des messages d’erreur retravaillés aux utilisateurs doivent pourvoir distinguer les différents problèmes métiers à partir du type de l’exception.

Les nouveaux types d’exception doivent respecter la convention de nommage suivante [TypeErreur]Exception et supporter le paramètre innerException.

#### Encapsulation d’une exception

Dans le cas où on souhaite catcher une exception pour lever une nouvelle exception différente, il faut impérativement renseigner le paramètre « innerException » de la nouvelle exception. Cette information est d’une grande importance pour le déboguage et la journalisation.

#### Gérer une exception et la transmettre

Dans le cas où l’on souhaite catcher une exception pour effectuer un traitement mais en laissant l’exception parvenir à l’utilisateur, il faut utiliser l’instruction throw; sans spécifier d’exception. Ceci permet de conserver la stack trace initiale après le block catch.

L’exemple suivant est une mauvaise pratique, car la stack trace initiale est perdue :

catch (ArticleInvalideException exception)  
{  
 this.LoguerException(exception);  
 throw exception ;  
}

L’exemple suivant est une gestion correcte de l’exception, car la stack trace initiale est conservée :

catch (ArticleInvalideException exception)  
{  
 this.LoguerException(exception);  
 throw;  
}

1. <https://martinfowler.com/bliki/BoundedContext.html> [↑](#footnote-ref-1)
2. <https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern> [↑](#footnote-ref-2)
3. <https://www.w3schools.com/xml/xml_soap.asp> [↑](#footnote-ref-3)
4. <http://www.castleproject.org/projects/dynamicproxy/> [↑](#footnote-ref-4)
5. <https://www.nunit.org/> [↑](#footnote-ref-5)
6. <https://github.com/moq/moq4/wiki> [↑](#footnote-ref-6)
7. <https://msdn.microsoft.com/fr-fr/library/dd901590(v=vs.95).aspx> [↑](#footnote-ref-7)
8. <https://msdn.microsoft.com/en-us/library/gg696172(v=vs.103).aspx> [↑](#footnote-ref-8)
9. <https://msdn.microsoft.com/fr-fr/library/bb386964(v=vs.110).aspx> [↑](#footnote-ref-9)
10. <http://www.markhneedham.com/blog/2009/07/07/domain-driven-design-anti-corruption-layer/> [↑](#footnote-ref-10)
11. <http://www.codemag.com/Article/0809101> [↑](#footnote-ref-11)
12. <https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/client-architecture> [↑](#footnote-ref-12)
13. <https://www.nuget.org/packages/NUnit/> [↑](#footnote-ref-13)
14. <https://www.nuget.org/packages/Moq/> [↑](#footnote-ref-14)
15. <https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions> [↑](#footnote-ref-15)