

Complément au Guide du Développeur

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Commentaire** | **Auteur** | **Société** |
| 0.1 | 27/09/2017 | Création du document. | Vincent JACQUOT | EID |
| 0.2 | 09/10/2017 | Ajout des normes de codage. | Vincent JACQUOT | EID |
| 0.3 | 11/10/2017 | Ajout des exemples de codes. | Vincent JACQUOT | EID |
| 0.4 | 12/10/2017 | Plus d’archivage des scripts SQL. | Vincent JACQUOT | EID |
| 1.0 | 20/10/2017 | Section sur les différents modèles de communication et exemples de code supplémentaires pour la gestion de l’asynchrone. | Vincent JACQUOT | EID |
| 1.1 | 25/10/2017 | * Normes pour l’utilisation des messages de commande. * Outils et normes pour l’implémentation des machines à états. * Corrections de fautes et de normes. | Vincent JACQUOT | EID |
| 1.2 | 03/11/2017 | * Précision de la norme des namespaces dans les projets des interfaces. | Vincent JACQUOT | EID |
| 1.3 | 06/11/2017 | * Ajout du paquet des algorithmes | Vincent JACQUOT | EID |
| 1.4 | 14/11/2017 | * Amélioration des schémas * Ajout d’un schéma pour l’envoi de commandes | Vincent JACQUOT | EID |
| 1.5 | 15/11/2017 | * Ajout du fournisseur d’identité. * Exemples de code sous forme de références vers TFS. | Vincent JACQUOT | EID |
| 1.5.1 | 11/12/2017 | * Correction d’une erreur dans la norme de codage des événements | Vincent JACQUOT | EID |

Table des matières

[1 Introduction 5](#_Toc500754362)

[2 Outils spécifiques 6](#_Toc500754363)

[2.1 Algorithmes 6](#_Toc500754364)

[2.1.1 Formule de Luhn 6](#_Toc500754365)

[2.2 Communication 6](#_Toc500754366)

[2.2.1 Client WCF 6](#_Toc500754367)

[2.3 Identification 6](#_Toc500754368)

[2.3.1 Identité 6](#_Toc500754369)

[2.3.2 Fournisseur d'identité 7](#_Toc500754370)

[2.4 Persistance 7](#_Toc500754371)

[2.4.1 Générateur de clés 7](#_Toc500754372)

[2.4.2 Générateur de séquences 8](#_Toc500754373)

[2.5 Entités 8](#_Toc500754374)

[2.5.1 Machine à états 8](#_Toc500754375)

[2.6 Paramétrage 8](#_Toc500754376)

[2.7 Tests 9](#_Toc500754377)

[2.7.1 Source de données 9](#_Toc500754378)

[2.7.2 Générateur de clés 9](#_Toc500754379)

[2.8 Initialisation 9](#_Toc500754380)

[2.9 Journalisation 10](#_Toc500754381)

[3 Communication entre domaines 11](#_Toc500754382)

[3.1 Communication entre domaines internes 11](#_Toc500754383)

[3.2 Communication avec un domaine externe 12](#_Toc500754384)

[3.2.1 Communication synchrone 12](#_Toc500754385)

[3.2.2 Communication asynchrone 13](#_Toc500754386)

[3.2.3 Modèles de communication courants 15](#_Toc500754387)

[4 Normes générales 18](#_Toc500754388)

[4.1 Team Foundation Server 18](#_Toc500754389)

[4.2 Compilateur 18](#_Toc500754390)

[4.3 Sonar 18](#_Toc500754391)

[4.4 Normes supplémentaires 19](#_Toc500754392)

[4.4.1 Organisation des fichiers 19](#_Toc500754393)

[4.4.2 Conventions de nommage 19](#_Toc500754394)

[4.4.3 Ordres de déclaration des membres d’un type 19](#_Toc500754395)

[4.4.4 Enumérations 20](#_Toc500754396)

[4.4.5 Lisibilité 20](#_Toc500754397)

[4.4.6 Usage des types 21](#_Toc500754398)

[4.4.7 Autres normes 21](#_Toc500754399)

[5 Checklist 22](#_Toc500754400)

[5.1 Structure du domaine 22](#_Toc500754401)

[5.2 Types communs 22](#_Toc500754402)

[5.2.1 Types énumérés 23](#_Toc500754403)

[5.2.2 Messages 23](#_Toc500754404)

[5.3 Code métier 24](#_Toc500754405)

[5.3.1 Entités 25](#_Toc500754406)

[5.3.2 Machines à états 26](#_Toc500754407)

[5.3.3 Interfaces des repositories 29](#_Toc500754408)

[5.3.4 Implémentation des repositories 29](#_Toc500754409)

[5.3.5 Interfaces des systèmes externes 30](#_Toc500754410)

[5.3.6 Interface du paramétrage métier 31](#_Toc500754411)

[5.3.7 Services métier 31](#_Toc500754412)

[5.4 Services applicatifs 32](#_Toc500754413)

[5.4.1 Objets de présentation 32](#_Toc500754414)

[5.4.2 Interfaces des services 33](#_Toc500754415)

[5.4.3 Implémentation des services 34](#_Toc500754416)

[5.4.4 Convertisseurs 36](#_Toc500754417)

[5.5 Tests unitaires 36](#_Toc500754418)

[5.6 Services externes 37](#_Toc500754419)

[5.6.1 Interface du paramétrage technique 37](#_Toc500754420)

[5.6.2 Implémentation du paramétrage 37](#_Toc500754421)

[5.6.3 Service applicatif d’un autre domaine interne 38](#_Toc500754422)

[5.6.4 Service applicatif synchrone d’un autre domaine externe 38](#_Toc500754423)

[5.6.5 Communication asynchrone par commandes avec un autre domaine 39](#_Toc500754424)

[5.6.6 Web Service avec une Web Reference 39](#_Toc500754425)

[5.7 Scripts SQL 40](#_Toc500754426)

[5.7.1 Tables de paramètres 42](#_Toc500754427)

[6 Questions et problèmes fréquents 44](#_Toc500754428)

# Introduction

Les briques du moteur du projet *EIT.Fixe* s’appuient sur le framework EIT. Le framework EIT est fourni avec un guide du développeur qui décrit les pratiques de développement et documente l’utilisation du framework.

Cependant, par rapport à l’utilisation standard du framework EIT, le développement des briques du moteur *EIT.Fixe* ajoute des contraintes spécifiques et met à disposition des outils supplémentaires.

Ce document a pour objectif d’apporter une description des pratiques, des contraintes et des outils de développements spécifiques au projet *EIT.Fixe* et qui n’ont pas leur place dans le guide du développeur standard du framework EIT.

La première section du document explique l’utilisation des outils supplémentaires communs aux briques du moteur *EIT.Fixe*. La deuxième section liste les conventions de codage adoptées dans le cadre du projet fixe. La troisième section fournit une liste des différents éléments à vérifier au cours du développement d’une brique du moteur *EIT.Fixe*. La quatrième et dernière section regroupe les questions et les problèmes fréquents.

Ce document s’appuie les exemples de code de la solution EIT.Fixe.Exemple récupérable depuis le dépôt du même nom.

# Outils spécifiques

Les différentes solutions qui composent le moteur *EIT.Fixe* s’appuient sur des outils communs qui surchargent le fonctionnement du framework EIT. Ces outils sont présents dans l’espace de noms **EIT.Fixe.Systeme**. Ils sont mis à disposition dans des paquets NuGet dont le nom est préfixé par EIT.Fixe.Systeme.

## Algorithmes

Le paquet EIT.Fixe.Systeme.Algorithmes contient des implémentations d’algorithmes courants qui sont accessibles depuis des méthodes statiques.

### Formule de Luhn

La formule de Luhn permet de vérifier la validité d’un identifiant composé de chiffres à l’aide d’un chiffre de contrôle. Il s’agit d’une forme simple de somme de contrôle. Le paquet des algorithmes propose une méthode qui permet de calculer le chiffre de contrôle à partir d’un identifiant.

## Communication

Le paquet EIT.Fixe.Systeme.Communication contient des outils qui aident à communiquer avec les autres briques du moteur ou avec des systèmes externes.

### Client WCF

Le paquet contient une fabrique de clients WCF qui simplifie l’instanciation des services. Elle permet d’instancier des services WCF à partir de l’interface de leur contrat et de l’URL du service. Les clients WCF obtenus doivent être disposés à la fin de leur utilisation.

Le fichier suivant contient un exemple de code montre comment instancier et utiliser un client WCF dans l’implémentation d’un système externe.

* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

## Identification

Le paquet EIT.Fixe.Systeme.Identification contient des outils qui permettent de collecter des informations sur l’identité de l’appelant d’un service applicatif.

### Identité

Le paquet définit un objet de présentation ***Identité*** qui contient des champs que l’appelant d’un service applicatif renseigne pour fournir son identité. Les services applicatifs destinés à être appelés en mode synchrone doivent demander un objet identité en premier paramètre de leurs méthodes. Les valeurs des champs de l’identité sont automatiquement enregistrées dans les journaux sur la ligne correspondante à l’appel de la méthode.

Les champs du paramètre identité sont les suivants :

* L’application appelante : c’est une chaîne qui identifie l’application qui a effectué l’appel. Dans le cas d’une IHM DevBooster ou d’un batch, on renseigne le nom du paquet ATEL. Dans le cas d’un domaine du moteur, on renseigne le nom complet du domaine.
* Le MEMOID : il s’agit du PERSOID de la personne qui a déclenché l’appel (si la donnée est disponible et pertinente).
* Le canal : le canal provenant de la liste des canaux.
* Le point de vente : on renseigne l’identifiant du point de vente qui a effectué l’appel (si la donnée est disponible et pertinente).

Les fichiers suivants contiennent un exemple de code qui montre comment utiliser le paramètre identité dans un service côté serveur et côté client.

* [IClientService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Interface%2FIClientService.cs&fullScreen=true)
* [ClientService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientService.cs&fullScreen=true)
* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

### Fournisseur d'identité

Le fournisseur d’identité est une interface définie dans le paquet EIT.Fixe.Systeme.Identification et qui permet à l’utilisateur de récupérer l’identité de l’appelant définie pour le contexte de l’appel. Dans le cas des méthodes des services applicatifs synchrones, le fournisseur d’identité utilise simplement l’identité passée en paramètre par l’appelant. Dans le cas des services asynchrones, une identité technique sera générée.

Les fichiers suivants contiennent un exemple de code qui montre comment utiliser le fournisseur d’identité dans une entité et comment le récupérer depuis un service applicatif.

* [ClientService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientService.cs&fullScreen=true)
* [Client.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain%2FClient.cs&fullScreen=true)

## Persistance

Les paquets EIT.Fixe.Systeme.Persistance et EIT.Fixe.Systeme.Persistance.Oracle contiennent des outils qui aident à gérer la persistance des entités. Le premier paquet contient des outils génériques et des interfaces alors que le second contient des implémentations propres à Oracle. Afin de garantir l’indépendance du code métier vis-à-vis des éléments techniques, seul le premier paquet peut être directement référencé.

### Générateur de clés

Le paquet EIT.Fixe.Systeme.Persistance définit l’interface d’un générateur de clés qui permet de générer des clés entières de type int ou long pour les entités. Le générateur prend en paramètre le type d’une entité et retourne une clé entière unique pour le type. L’implémentation du générateur de clé s’appuie des séquences en base de données.

Les fichiers suivants contiennent un exemple de code qui montre comment utiliser le générateur de clés avec une base Oracle dans un service applicatif et dans un service métier.

* [ArticleService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application%2FArticleService.cs&fullScreen=true)
* [GestionnaireCommandes.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FGestionnaireCommandes.cs&fullScreen=true)

### Générateur de séquences

Le paquet EIT.Fixe.Systeme.Persistance définit l’interface d’un générateur de séquences qui utilise la base de données et sur lequel s’appuie le générateur de clés. Il permet d’obtenir des entiers de type int ou long à partir du nom de la séquence dans la base de données. L’utilisation directe du générateur de séquence est à éviter car ceci couple le code à la base de données. Il est préférable d’utiliser le générateur de clés quand cela est possible.

## Entités

Le paquet EIT.Fixe.Systeme.Entites contient des outils qui aident à modéliser les entités. Ce paquet peut être référencé depuis les projets de code métier du domaine.

### Machine à états

Le paquet EIT.Fixe.Systeme.Entites contient des outils qui permettent d’implémenter le patron de conception *état* sur des entités persistantes. Il fournit une classe base dont doivent dériver les états de l’entité et une classe de base pour le type de la propriété qui permet de stocker l’état courant de l’entité.

Les fichiers suivants contiennent un exemple de code qui montre comment utiliser les outils du paquet EIT.Fixe.Systeme.Entites pour implémenter le patron de conception *état*.

* [Commande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.cs&fullScreen=true)
* [EtatCommande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FEtatCommande.cs&fullScreen=true)
* [Commande.AbstractEtat.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.AbstractEtat.cs&fullScreen=true)
* [Commande.EtatEnAttenteValidation.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.EtatEnAttenteValidation.cs&fullScreen=true)
* [Commande.EtatValidee.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.EtatValidee.cs&fullScreen=true)
* [Commande.MachineEtats.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.MachineEtats.cs&fullScreen=true)

## Paramétrage

Le paquet EIT.Fixe.Systeme.Parametrage.Oracle contient des outils qui aident à la lecture de tables de paramétrage depuis une base Oracle.

Il définit une classe qui permet de récupérer du paramétrage depuis une table dans une base de données Oracle. Elle gère automatiquement la mise en cache des paramètres pour 5 minutes. L’utilisateur doit dériver de la classe et renseigner le nom de la table ainsi qu’un fournisseur de connexion Oracle. La table de paramétrage en base doit respecter un formalisme pour le nom et le type de ses colonnes.

Le fichier suivant contient un exemple de code qui montre comment utiliser le lecteur de table de paramétrage avec une base Oracle.

* [Parametrage.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FParametrage.cs&fullScreen=true)

## Tests

Le paquet EIT.Fixe.Systeme.Tests contient des classes utilitaires qui aident à réaliser les tests unitaires. En particulier, elle propose des implémentations en mémoire de classes de persistances qui permettent de lancer des tests sans impacter la base de données.

### Source de données

Le paquet EIT.Fixe.Systeme.Tests contient une implémentation en mémoire d’une source de données utilisable par un registre. Elle permet de réaliser des tests sur les registres sans avoir besoin d’une base de données réelle.

Le fichier suivant contient un exemple de code qui montre comment utiliser la source de données en mémoire pour tester un registre.

* [ClientRepositoryTest.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain.Test%2FRepositories%2FClientRepositoryTest.cs&fullScreen=true)

### Générateur de clés

Le paquet EIT.Fixe.Systeme.Tests contient une implémentation en mémoire du générateur de clés. Elle permet de réaliser des tests de création d’entités par les services applicatifs ou les services métier sans avoir besoin d’une base de données réelle.

Le fichier suivant contient un exemple de code qui montre comment utiliser le générateur de clés en mémoire pour tester la création d’entités.

* [ClientServiceTest.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Test%2FClientServiceTest.cs&fullScreen=true)

## Initialisation

Le paquet EIT.Fixe.Systeme.Initialisation contient des outils qui aident à la configuration et à l’initialisation de l’application. C’est un paquet technique qui est très spécifique au projet *EIT.Fixe*.

Il met à disposition une classe d’application globale abstraite qui configure les éléments suivants :

* Le fournisseur de connexion pour la base de données Oracle
* L’export des modules techniques et utilitaires
* L’export automatique des registres
* L’export et l’exposition automatique des services

L’export automatique des registres et des services repose sur le nom des assemblages qui les contiennent. Les assemblages pris en compte par l’export automatique doivent être nommés avec un préfixe commun. La classe met également à disposition une méthode protégée qui permet de faire manuellement l’export des modules qui ne seraient pas automatiquement exportés.

La classe d’application globale prend en paramètre le nom de la base de données ciblée par l’application et un booléen qui permet d’activer le mode de débogage. Le mode de débogage permet d’afficher les requêtes générées par l’ORM sur la sortie standard. Il ne doit pas être activé en production.

Le fichier suivant contient un exemple de code qui montre comment utiliser la classe d’application globale.

* [Global.asax.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Infrastructure.WebApplication%2FGlobal.asax.cs&fullScreen=true)

## Journalisation

Le paquet EIT.Fixe.Systeme.Journalisation contient un module de journalisation en base de données pour le framework EIT.

Le module de journalisation en base de données est automatiquement configuré et exporté par le paquet EIT.Fixe.Systeme.Initialisation. Il permet d’enregistrer une trace de chaque appel d’un service applicatif dans une base de données Oracle. La trace contient, entre autres :

* La date du début de l’appel
* Le nom du service
* Le nom de la méthode
* Les champs de l’identité fournie par l’appelant
* Les paramètres passés à la méthode
* Le temps d’exécution
* En cas d’erreur, le message de l’exception remontée
* En cas d’erreur, la pile d’exécution

# Communication entre domaines

Le moteur du projet EIT.Fixe est une application unique composée de multiples domaines internes. Les domaines internes peuvent communiquer entre eux mais également communiquer avec des domaines externes. Cette section explique comment les différents domaines communiquent entre eux.

## Communication entre domaines internes

Les domaines internes sont les domaines préfixé par EIT.Fixe qui appartiennent au même dépôt Git, à la même solution EIT.Fixe et qui sont exécutés dans la même application EIT.Fixe. Ils partagent évidemment la même base de donnés.

Les domaines se considèrent entre eux comme des systèmes externes et chaque code métier est indépendant. L’implémentation du système externe communique avec l’autre domaine les interfaces de ses services applicatifs, c’est-à-dire les mêmes interfaces que celles qui sont exposées sous forme de services WCF.

Cependant, contrairement à un domaine externe, les services applicatifs des différents domaines internes partagent une même transaction. Ainsi, si l’un des domaines lève une erreur qui remonte jusqu’au service applicatif initialement appelé, un retour arrière sera effectué sur l’intégralité des données modifiées.

Le schéma suivant illustre la communication synchrone entre deux domaines internes au sein de l’application EIT.Fixe.



Le fichier suivant contient un exemple de code qui montre comment communiquer avec un autre domaine interne à travers un service externe.

* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

## Communication avec un domaine externe

La communication avec un domaine externe passe nécessairement par le réseau. Il existe deux modes de communication possible : un mode communication synchrones via des Web Services ou un mode de communication asynchrone via des messages qui transitent sur le broker Kafka.

### Communication synchrone

La communication synchrone avec un domaine externe s’effectue à l’aide de Web Service exposés par soit par le domaine de l’application EIT.Fixe, soit par le domaine externe. En fonction de la technologie supportée par le domaine externe, le Web Service sera consommé soit via des interfaces WCF, soit avec une Web Reference générée depuis un fichier WSDL.

La communication synchrone permet à l’appelant d’attendre le résultat de l’appel avant de poursuivre son traitement. Cependant, ce mode de communication n’est pas transactionnel. Ainsi, si un appel synchrone effectue des modifications de données dans le domaine appelé, et qu’une erreur survient dans le domaine appelant, aucun retour arrière ne sera effectué automatiquement sur les données du domaine appelé. L’intégrité des données est donc compromise dans certains cas d’erreur.

La communication synchrone par Web Service est donc adaptée pour les appels de consultation en lecture seule. L’interaction avec des interfaces utilisateur et la réservation de ressource oriente également vers une communication synchrone si des précautions sont prises pour résoudre les problèmes d’intégrité des données. Les traitements des appels synchrones doivent également être rapides car l’appel est bloquant dans le flux de contrôle de l’appelant.

Le schéma suivant illustre la communication par Web Service entre deux domaines.



Le fichier suivant contient un exemple de code qui montre comment communiquer par Web Service avec un domaine externe.

* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

### Communication asynchrone

La communication asynchrone avec un domaine externe s’effectue à l’aide de messages qui transitent sur le broker Kafka. Les messages peuvent également être employés pour déclencher des traitements asynchrones dans le même domaine ou dans un autre domaine interne. Le domaine qui émet le message est appelé producteur et celui qui reçoit le message consommateur. Il peut exister plusieurs consommateurs ou plusieurs producteurs différents pour un même type de message.

L’émission des messages est faite de manière transactionnelle. C’est-à-dire qu’en cas d’erreur dans la transaction du producteur, le message ne sera pas envoyé au broker. L’intégrité des données est donc garantie à terme si le consommateur traite correctement le message.

La communication asynchrone est donc adaptée pour réaliser des modifications de données ou pour confirmer des modifications de données. Elle est également utile pour déclencher des traitements qui prennent du temps chez le consommateur car le traitement du message n’est pas bloquant dans le flux de contrôle du producteur.

Le schéma suivant illustre la communication par messages de type *événement* entre deux domaines. Les domaines représentés sont dans des applications différentes mais la communication asynchrone entre deux domaines internes est également possible.



Les fichiers suivants contiennent un exemple de code qui montre comment définir le format des messages, produire des messages dans le code métier (message de type *évènement*) et consommer des messages dans la couche application.

* [CommandeCreeEvent.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain.CommonTypes%2FEvents%2FCommandeCreeEvent.cs&fullScreen=true)
* [Commande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.cs&fullScreen=true)
* [IClientEventService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Interface%2FIClientEventService.cs&fullScreen=true)
* [ClientEventService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientEventService.cs&fullScreen=true)

Le schéma suivant illustre la communication par messages de type *commande* entre deux domaines dans des applications différentes.



Les fichiers suivants contiennent un exemple de code qui montre comment définir le format des messages, produire des messages dans le code des services externes (message de type *commande*) et consommer des messages dans la couche application.

* [AvertissementClientCommand.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain.CommonTypes%2FCommands%2FAvertissementClientCommand.cs&fullScreen=true)
* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)
* [IClientEventService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Interface%2FIClientEventService.cs&fullScreen=true)
* [ClientEventService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientEventService.cs&fullScreen=true)

### Modèles de communication courants

Les appels de services synchrones et les échanges de messages asynchrones sont des outils techniques de communication élémentaires. Des modèles de communication plus conceptuels s’appuient sur ces outils élémentaires. Les contraintes fonctionnelles ou techniques nécessitent parfois de mettre en place des modèles de communication plus complexes qui vont combiner les deux modes de fonctionnement.

#### Evènement

Un évènement est un message asynchrone qui contient des informations mises à disposition de consommateurs à priori inconnus par le producteur. Le producteur émet l’évènement de manière transactionnelle et ne souci pas des traitements effectués par les consommateurs.

Le format du message de l’évènement est défini par le **producteur**. Ainsi, ce modèle est adapté lorsque les producteurs sont des applications indépendantes qui ne connaissent pas le métier ou la responsabilité des consommateurs potentiels du message.

Techniquement, l’évènement est émis depuis la couche *Domain* du producteur.

#### Commande

Une commande est un message asynchrone qui contient un ordre à destination d’un seul consommateur précis. Le producteur émet la commande de manière transactionnelle et n’attend pas de retour immédiat de la part du consommateur.

Le format du message de la commande est défini par le **consommateur**. Ainsi, ce modèle est adapté lorsque les consommateurs sont des applications de type « boite à outils » génériques qui ne connaissent pas le métier des producteurs.

Techniquement, l’évènement est émis depuis la couche *Services Externes* du producteur.

#### Validation asynchrone

Les appels synchrones d’écriture entre deux applications sont parfois nécessaires lorsqu’un retour immédiat est attendu dans une interface cliente. Malheureusement, comme les appels synchrones ne garantissent pas l’intégrité transactionnelle, des problèmes d’incohérence de données entre les deux applications peuvent apparaître. En particulier, l’application appelante peut rencontrer une erreur après avoir effectué son appel synchrone et ainsi ne pas garder la trace du bon traitement de l’appel par l’application appelée.

Pour se protéger contre ces incohérences, l’application appelée peut imposer une procédure de validation asynchrone après les appels synchrones. La méthode d’écriture synchrone fournira un retour immédiat à l’appelant mais laissera initialement les données créées dans un état temporaire. Pour valider définitivement les nouvelles données, l’application appelée attend la réception d’un message de validation.

L’envoi du message de validation est effectué par l’appelant immédiatement après l’appel synchrone d’écriture. L’envoi des messages étant transactionnel, le message ne sera pas envoyé si l’application appelante rencontre une erreur et les données resteront en attente de validation dans l’application appelée. Les données non validées dans l’application appelée peuvent être soit ignorées, soit traitées en support ou par un batch s’il faut libérer des ressources associées.

Le schéma ci-dessous illustre le fonctionnement de ce modèle de validation asynchrone.



# Normes générales

Les développements dans le projet *EIT.Fixe* doivent respecter des normes de codage. La mise en place des conventions de codage a de multiples objectifs :

* Favoriser la lisibilité du code.
* Standardiser le style de codage pour faciliter la relecture.
* Eviter les structures qui induisent des problèmes de performance.

La vérification des conventions est partiellement effectuée par le compilateur et l’outillage mais repose également sur la vigilance des développeurs.

## Team Foundation Server

Team Foundation Server sert de plateforme d’intégration continue, de déploiement et de gestion du code source. Le gestionnaire de code source sous-jacent est Git.

Les développements doivent s’effectuer sur la branche *develop* ou une branche de *feature* qui dérive de la branche *develop*. Le code est ensuite rapatrié sur la branche *master* via des processus de *pull request*. Ceci permet d’effectuer une validation et une relecture du code avant d’accepter le code dans la branche principale.

Tous les *commits* de code sur le dépôt Git doivent être associés à un *work item* de type *feature*. Les *work items* doivent être nommés selon la fonctionnalité correspondante des spécifications. Le nommage comporte le code de la fonctionnalité sous forme de trigrammes et le nom de la fonctionnalité qui débute par un verbe à l’infinitif comme suit : **[CodeFonctionnalité] – Nom fonctionnalité**.

## Compilateur

Les développeurs doivent corriger les avertissements (*warnings*) remontés par le compilateur MSBuild incorporé dans Visual Studio.

## Sonar

***Sonar*** est un outil d’analyse statique qui permet de vérifier que le code vérifier une liste de règles. L’outil se décline en deux variantes :

* Le serveur ***SonarQube*** effectue des analyses distantes sur une branche du dépôt déclenchées par l’intégration continue. Il permet d’effectuer un suivi du nombre de règles enfreintes et permet d’affecter les problèmes aux développeurs pour résolution.
* ***SonarLint*** est un plugin pour Visual Studio qui communique avec un serveur SonarQube pour synchroniser une liste de règles. Il remonte directement les règles enfreintes dans Visual Studio sous forme d’avertissements ou d’erreurs de compilation. Son utilisation est facultative.

Les développeurs doivent effectuer un suivi régulier du nombre de règles enfreintes dans leur code sur le serveur *SonarQube*. Les problèmes remontés par l’outil doivent être corrigés.

## Normes supplémentaires

Des normes de codage supplémentaires non matérialisées par des avertissements du compilateur ou des règles Sonar sont en application. Certaines règles qui s’appliquent à des éléments de code spécifiques sont listées dans la section [Checklist](#_Guide_synthétique). D’autres dont le caractère est plus général sont listées ci-dessous.

### Organisation des fichiers

Les fichiers ne doivent contenir qu’un seul type (interface, classe, énumération, structure…). Le nom du fichier doit alors correspondre au nom du type qu’il contient. Les fichiers doivent respecter l’arborescence standard des projets décrite dans la section [Checklist](#_Guide_synthétique).

### Conventions de nommage

Les noms des symboles (types, membres, variables…) sont en français afin de correspondre au plus près au langage du métier et des spécifications. Ils doivent également, d’une manière générale, respecter les conventions préconisées par le MSDN ainsi que les règles internes suivantes :

* Les noms des symboles (types, membres, variables…) ne doivent contenir que des caractères alphanumériques (par de caractères accentués) et les chiffres sont à éviter au maximum.
* Le nom des interfaces débute par un « I » majuscule.
* Le nom des méthodes commence par un verbe à l’infinitif.
* Les propriétés booléenne comment par « Est » ou « A ».
* Les énumérateurs (valeurs d’énumération) commencent par une majuscule.
* Le nom des constantes est en majuscule, les mots séparés par des « \_ » (MA\_CONSTANTE).
* Le nom des Web Methods doit rester standard et cohérent dans l’application en favorisant l’usage de certains mots clés standards : Rechercher, Obtenir, Lister, Definir…

### Ordres de déclaration des membres d’un type

L’ordre des membres d’un type est standardisé afin de faciliter la relecture du code. Ainsi, dans une classe, une structure ou une interface, on doit trouver, dans l’ordre :

1. Les constantes (toujours privées ou internes)
2. Les champs (toujours privés)
3. Les propriétés
4. Les constructeurs
5. Les méthodes
6. Les évènements (au sens .NET)
7. Les membres qui implémentent des interfaces, groupés par interface

A l’intérieur de chaque bloc, les membres sont triés par visibilité :

1. Les membres privés
2. Les membres protégés
3. Les membres internes
4. Les membres publics

Les membres qui implémentent des interfaces sont groupés à la fin du type dans des régions qui portent le nom des interfaces qui sont implémentées. Si l’implémentation d’une interface est trop volumineuse, elle peut être externalisée dans un autre fichier à l’aide du modificateur partial. Le fichier qui contient l’implémentation est alors nommé [NomType].[NomInterface].

### Enumérations

Les normes suivantes concernent les types énumérés. Ces derniers ont un comportement un peu particulier, notamment à cause des valeurs sous-jacentes des énumérateurs.

* Les énumérateurs (valeurs de l’énumération) doivent avoir une valeur numérique explicite. En effet, si la valeur sous-jacente n’est pas renseignée, elle est déterminée à partir de l’ordre de déclaration des énumérations, ce qui rend le code fragile.
* Il doit toujours exister un énumérateur correspondant à la valeur « 0 » car il s’agit de la valeur par défaut, même si aucun énumérateur n’existe.
* La valeur par défaut ne doit pas avoir de signification métier, afin d’éviter de prendre un compte dans un processus métier une valeur qu’on a oublié d’initialiser. L’énumérateur correspondant à la valeur par défaut doit être nommée NA.
* L’ordre des énumérations d’une énumération qui représente les états d’une machine à états d’un objet doit suivre autant que possible le cycle de vie normal de l’objet. Ceci facilite la lisibilité. Les valeurs des énumérateurs peuvent être affectées de 10 et 10 afin anticiper l’ajout d’étapes intermédiaires.

### Lisibilité

Les conventions suivantes doivent être respectées afin de favoriser la lisibilité du code.

* Les directives #region et #endregion sont toujours nommées, afin d’indiquer quelle est la région qui débute ou qui s’achève.
* Les régions sont proscrites à l’intérieur des méthodes pour éviter de compliquer la lecture en dissimulant de gros blocs de code.
* L’usage des membres d’instance est toujours préfixé par le mot clé this.
* L’usage des membres statiques est toujours préfixé 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.
* Tous les types publics disposent d’une documentation XML.
* On évite d’utiliser un constructeur de délégué quand on peut utiliser un nom de method group.
* Le mot clé var n’est pas utilisé sauf dans le cas d’une construction d’objet avec un new (le type est directement visible) ou dans une boucle foreach (pour éviter les casts malheureux).
* La taille des méthodes est, dans la mesure du possible, contenue à quelques dizaines de lignes pour qu’elles restent lisibles.
* L’usage d’opérateurs ternaires ne doit pas nuire à la lisibilité, il convient en particulier d’éviter les opérateurs ternaires imbriqués.
* Les montants doivent être exprimés en euro afin d’éviter toute confusion.

### Usage des types

Les normes suivantes concernent l’usage des types.

* L’usage du type double est proscrit afin d’éviter les problèmes de précision, en particulier sur les montants. Il faut lui préférer le type decimal.
* Les variables d’itération dans les boucles foreach ne doivent pas être typées manuellement mais doivent utiliser le mot clé var. En effet, dans ce cas précis, le compilateur ne lève pas d’erreur si le typage est incorrect et un problème de cast peut apparaître à l’exécution.
* Les types sbyte, ushort, uint, ulong, qui ne sont pas CLS-compliants, ne doivent pas être utilisés.

### Autres normes

Les normes suivantes sont des bonnes pratiques de modélisation ou de codage pour éviter les erreurs.

* Le multithreading explicite est interdit, sauf l’usage des méthodes asynchrones avec les mots clés async et await.
* Les collections membres d’une classe sont initialisées dans le constructeur de la classe. On doit d’une manière générale préférer les collections vides aux collections nulles.
* Les objets qui possèdent une notion d’état doivent implémenter le design pattern de la machine à états.
* Les tests sur des valeurs de clés « en dur » sont à proscrire.

# Checklist

Cette section propose un guide de développement sous la forme de listes de points à contrôler à chaque étape du développement. La checklist permet de s’assurer du respect de normes de codage spécifiques, des bonnes pratiques de développement, et de la bonne utilisation du framework pour éviter certaines erreurs courantes.

## Structure du domaine

Au cours du développement, il faut s’assurer que les projets qui composent les domaines ont le bon contenu et les bonnes dépendances. En particulier, le code métier doit être indépendant des briques techniques et il ne faut pas contourner des interfaces pour utiliser directement une implémentation.

Le schéma ci-dessous sert de référence pour le contenu et les dépendances des projets d’un domaine. Les projets de tests unitaires ne sont pas représentés.



## Types communs

Les types communs sont des types ou des objets valeurs non persistés qui sont spécifiques au domaine mais qui peuvent apparaître dans les différentes couches. Les types communs sont utilisables dans les couches du moteur mais n’ont pas vocation à être utilisés dans les IHM. Lors de l’écriture des types communs, les points suivants sont à vérifier :

* **Les types communs se trouvent dans le projet EIT.Fixe.[Domaine].Domain.CommonTypes.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets des types communs par les outils.

### Types énumérés

Les types énumérés transverses ont leur place de projet des types communs. Lors de l’écriture de ces types énumérés, les points suivants sont à vérifier :

* **Les types énumérés sont placés dans le répertoire Enumerations.**Cette norme organise les sources et facilite la relecture du code.
* **Les valeurs numériques des énumérateurs sont explicitement renseignées.**Les valeurs numériques sont potentiellement utilisées par l’ORM pour la persistance. Il faut donc les fixer pour éviter les accidents en cas de modification de la liste des énumérateurs.
* **Il existe toujours un énumérateur correspondant à la valeur 0.**Définir un énumérateur pour la valeur zéro évite des erreurs à l’exécution et permet de travailler avec la valeur par défaut sans avoir à *caster* l’énumération dans son type sous-jacent.
* **L’énumérateur correspondant à la valeur zéro n’est pas significatif pour le métier.**Cette norme permet d’éviter de prendre en compte à tort dans le métier des cas d’erreur ou la valeur d’énumération n’est pas initialisée. La valeur zéro doit correspondre à un énumérateur nommé NA.

Le fichier suivant contient un exemple de code qui montre comment écrire un type énuméré commun.

* [Civilite.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain.CommonTypes%2FEnumerations%2FCivilite.cs&fullScreen=true)

### Messages

Les messages asynchrones qui peuvent être véhiculés à travers le broker sont définis avec les types communs. Ils doivent être sérialisables. Lors de l’écriture de ces types énumérés, les points suivants sont à vérifier :

* **Le message se trouve dans le projet EIT.Fixe.[Domaine].Domain.CommonTypes.**Le message est un type transverse qui peut circuler du projet métier jusqu’aux interfaces des services applicatifs. Le respect de cette norme de nommage facilite également la relecture du code.
* **Les messages de type *évènement* se trouvent dans le répertoire Events.**Cette norme facilite la relecture du code.
* **Les messages de type *commande* se trouvent dans le répertoire Commands.**Cette norme facilite la relecture du code.
* **Les messages de type *évènement* sont nommés [NomEvenement]Event.**Cette norme de nommage facilite la relecture du code.
* **Les messages de type commande sont nommés [NomCommande]Command.**Cette norme de nommage facilite la relecture du code.
* **Le message est une classe**.  
  Il s’agit d’une contrainte de la sérialisation.
* **Le message, qu’il soit un évènement ou une commande, dérive du type Event.**  
  Cette norme est un prérequis pour le fonctionnement du système de communication asynchrone.
* **Le message possède un constructeur sans paramètre.**Le constructeur sans paramètre est nécessaire pour désérialiser l’objet de manière efficace.
* **Les propriétés sérialisables déclarent un accesseur get et un accesseur set.**Le sérialiseur doit pouvoir accéder aux propriétés sérialisables en lecture pour la sérialisation et en écriture pour la désérialisation. Les propriétés à qui il manque l’un des accesseurs ne sont pas sérialisées ou désérialisées.
* **Les propriétés sérialisables sont décorées avec l’attribut DataMember.**  
  Cet attribut permet d’indiquer au framework que la propriété de l’objet de présentation est sérialisable.
* **Les types des propriétés sérialisables font partie de la liste des types supportés.**  
  Le type des propriétés sérialisable doit faire partie de la liste des types suivants pour être supporté par le framework :
  + byte, bool, short, int, long, decimal ou DateTime
  + Un type énuméré
  + Nullable<T> ou T est l’un des types précédents
  + Un objet de présentation
  + Un tableau de l’un des types précédents

Le fichier suivant contient un exemple de code qui monte comment écrire un évènement.

* [CommandeCreeEvent.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain.CommonTypes%2FEvents%2FCommandeCreeEvent.cs&fullScreen=true)

Le fichier suivant contient un exemple de code qui montre comment écrire une commande.

* [AvertissementClientCommand.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain.CommonTypes%2FCommands%2FAvertissementClientCommand.cs&fullScreen=true)

## Code métier

L’écriture du code métier est la première étape du développement. Elle concerne l’écriture des entités, des repositories, des services « métier » et les interfaces du paramétrage et des systèmes externes. En développant du code métier, les points suivants sont à vérifier :

* **Le code métier se trouve dans le projet EIT.Fixe.[Domaine].Domain.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets métier par les outils d’intégration continue et de configuration.
* **Les tests se trouvent dans le projet EIT.Fixe.[Domaine].Domain.Test.**Le respect de cette norme de nommage facilite la relecture du code et l’identification du projet de test par les outils d’intégration continue.

### Entités

Les entités sont des objets métier persistés qui sont identifiés par une clé. Deux entités sont identiques si et seulement si leurs clés sont identiques. Lors de l’écriture d’une entité, les points suivants sont à vérifier :

* **L’entité est une classe virtuelle.**  
  Ceci est nécessaire pour le fonctionnement de l’ORM et pour créer des objets *Mock* dans les tests unitaires.
* **L’entité dérive de la classe de base Entity.**  
  Cette classe de base est nécessaire au fonctionnement du système d’évènements.
* **L’entité concrète possède une classe de tests unitaires dans le projet de test du domaine.**  
  Toute entité concrète doit posséder une classe de tests unitaires. C’est un prérequis indispensable au développement selon l’approche TDD.
* **La classe de test de l’entité est nommée [Entite]Test.**Le respect de la norme de nommage facilite la relecture du code et l’identification des classes de test par les outils.
* **Les tests des membres publics de l’entité sont écrits avant leur implémentation.**  
  Il s’agit du principe de l’approche TDD.
* **Tous les membres publics ou protégés de l’entité sont virtuels.**  
  Ceci est indispensable pour pouvoir créer des objets *Mock* dans les tests unitaires. De plus, la fonctionnalité de *Lazy Loading* des ORM ne fonctionne qu’avec des membres virtuels.
* **L’entité est décorée avec l’attribut CustomTableName.**Cet attribut permet de renseigner le nom de la table qui correspond à l’entité en base en de données.
* **L’entité possède un constructeur protégé sans paramètre.**  
  Le constructeur sans paramètre est nécessaire pour que l’ORM puisse désérialiser l’entité. Il ne doit cependant pas être public pour éviter la création d’entités incohérentes ou sans clé.
* **Les propriétés persistantes déclarent un accesseur get et un accesseur set.**L’ORM doit pouvoir accéder aux propriétés persistantes en lecture pour la sérialisation et en écriture pour la désérialisation. Les propriétés à qui il manque l’un des accesseurs ne sont pas persistées.
* **Les types des propriétés persistantes font partie de la liste des types supportés.**Le type des propriétés persistantes doit faire partie de la liste suivante pour être supporté par le framework :
  + bool, short, int, long, decimal, string ou DateTime
  + Un type énuméré
  + Nullable<T> où T est l’un des types précédent
  + Un objet valeur
  + Une entité
  + ICollection<T> où T est une entité
* **Les propriétés persistantes sont décorées avec l’attribut CustomColumnName.**Cet attribut permet de renseigner le nom de la colonne
* **Les collections persistantes sont décorées avec l’attribut CustomAssociationTable.**Cet attribut permet de renseigner le nom de la table d’association et des colonnes qui correspondent à la collection en base de données.
* **L’entité possède une unique propriété décorée avec l’attribut Key nommée Cle.**L’attribut Key permet à la fois d’indiquer au framework que la classe est une entité et d’identifier l’unique clé de l’entité. La norme de nommage de la clé facilite la relecture du code.
* **L’entité fille qui dérive d’une entité mère ne redéfinit pas de clé.**  
  La clé est unique pour toute une hiérarchie d’entités. Elle doit être définie uniquement dans la classe mère.
* **Le premier paramètre des constructeurs publics est la clé de l’entité.**  
  La clé de l’entité peut être exploitée par le code métier et doit être connue dès sa création, et non lors de la sérialisation. De plus, il est préférable pour la testabilité que l’entité ne génère pas sa propre clé. Le placement systématique du paramètre de la clé en première position facilite la relecture du code.
* **Les évènements sont envoyés par l’intermédiaire d’un event de type Action<TypeEvenement>.**Le mécanisme d’envoi des messages de type *évènement* utilise un event (au sens .NET) défini sur l’entité qui lève l’évènement. Le type de l’event doit être Action<TypeEvenement> ou TypeEvenement est le type du message à envoyer. Cette norme est nécessaire au bon fonctionnement du système de gestion des messages. L’évènement sera automatiquement détecté par la classe base Entity comme étant un point d’entrée pour la création de messages.

Les fichiers suivant contiennent des exemples de code qui montrent comment écrire des entités.

* [Article.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FArticle.cs&fullScreen=true)
* [Commande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.cs&fullScreen=true)
* [LigneCommande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FLigneCommande.cs&fullScreen=true)

### Machines à états

Les différentes classes qui permettent d’implémenter le patron de conception *état* sur les entités sont logiquement placées dans le code métier. Lors de l’écriture d’un patron de conception *état*, les points suivants sont à vérifier :

* **Toute entité qui possède des états doit implémenter le patron de conception *état***.  
  Ce patron de conception permet de gérer le complexité du code des entités qui se comportent comme des machines à états ou dont le comportement diffère en fonction de leur état.
* **L’énumération Etat[NomEntite] liste les différents états possibles de l’entité.**  
  Cette énumération représente l’état de l’entité tel qu’il sera stocké en base de données.
* **Les états de l’énumération sont dans l’ordre logique du cycle de vie de l’entité.**  
  Cette norme facilite la lecture des données pour le support. Cette norme peut être sans objets pour certains états difficiles à placer chronologiquement dans le cycle de vie de l’entité.
* **Les valeurs des états de l’énumération sont numérotées de 10 et 10.**  
  Ceci permet de prévoir la place pour ajouter de nouveaux états entre deux étapes du cycle de vie de l’entité en cas de complexification du modèle.
* **Les classes du patron de conception état sont imbriquées dans la classe de l’entité.**  
  Cette norme standardise l’organisation du code et permet aux classes des états d’accéder aux propriétés privées et protégées de l’entité.
* **Une classe MachineEtats dérive de la classe MachineEtats<TEntite, TEnumeration, TEtat>**Cette classe permet de stocker l’état courant de l’entité. La classe base générique MachineEtats est la solution standardisée du projet *EIT.Fixe* pour gérer l’état courant d’une entité persistante. Elle gère les problèmes de sérialisation et de persistance de l’état.
* **La classe MachineEtats possède un constructeur protégé sans paramètre.**Le constructeur sans paramètre est nécessaire pour que l’ORM puisse désérialiser le type. Il ne doit cependant pas être public pour éviter la création d’objets incohérents.
* **La classe MachineEtats possède un constructeur public qui prend l’état initial en argument.**Le constructeur public doit permettre d’instancier la machine à états avec l’état initiale de l’entité. Il doit appeller le constructeur de la classe de base qui prend un état en paramètre.
* **Une classe abstraite AbstractEtat dérive de Etat<TEntite, TEnumeration>.**Cette classe de base abstraite représente le type d’état propre à l’entité. La classe de base générique Etat est la solution standardisée du projet *EIT.Fixe* pour gérer l’état d’une entité.
* **La classe de base des états possède un constructeur public qui prend l’entité en paramètre.**Selon le patron de conception, les états possèdent une référence vers l’entité. Cette référence doit être initialisée à la création de tous les états. Ce constructeur appelle le constructeur de la classe de base qui prend une entité en paramètre.
* **La classe de base des états définit les méthodes virtuelles des états.**Les méthodes des états doivent être définies sur la classe de base abstraite. Elles doivent être virtuelles pour que les états puissent les réimplémenter.
* **L’implémentation des méthodes virtuelles de la classe de base lèvent une exception.**Les méthodes des états doivent être réimplémentées par les états concrets lorsqu’elles sont valides. Lorsque la méthode n’est pas réimplémentée, elle doit lever une exception qui indique que l’action n’est pas valide sur l’état courant.
* **Le nom des états concrets est préfixé par Etat.**Cette norme de nommage facilite la relecture du code.
* **Les états concrets possèdent un constructeur public qui prend l’entité en paramètre.**Selon le patron de conception, les états possèdent une référence vers l’entité. Cette référence doit être initialisée à la création de tous les états. Ce constructeur appelle le constructeur de la classe de base qui prend une entité en paramètre.
* **Les états concrets doivent implémenter la propriété Valeur et retourner le bon état.**Cette norme est nécessaire au bon fonctionnement de la persistance des états. Elle permet d’associer l’énumérateur d’un état avec la classe correspondante.
* **Les états concrets doivent réimplémenter les méthodes qu’ils supportent.**Les différentes méthodes virtuelles définies sur la classe de base doivent être réimplémentées dans les états concrets qui les supportent.
* **L’entité possède une propriété persistée Etat de type MachineEtats.**  
  Cette propriété permet de persister l’état courant de l’entité.
* **La propriété Etat doit être initialisée à la construction de l’entité.**  
  Cette étape est indispensable pour initialise l’état de l’entité.
* **L’entité récupère l’état courant à l’aide d’une propriété EtatCourant.**Cette méthode facilite la lisibilité. Les accesseurs get et set utilisent respectivement les méthodes Obtenir() et Definir() de la classe MachineEtats pour retourner ou affecter la valeur courante de l’état.

Les fichiers suivants contiennent un exemple de code qui montre comment implémenter le patron de conception état sur une entité.

* [Commande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.cs&fullScreen=true)
* [EtatCommande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FEtatCommande.cs&fullScreen=true)
* [Commande.AbstractEtat.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.AbstractEtat.cs&fullScreen=true)
* [Commande.EtatEnAttenteValidation.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.EtatEnAttenteValidation.cs&fullScreen=true)
* [Commande.EtatValidee.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.EtatValidee.cs&fullScreen=true)
* [Commande.MachineEtats.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FCommande.MachineEtats.cs&fullScreen=true)

Le schéma suivant illustre les relations entre les différentes classes qui composent l’implémentation du patron de conception *état*.



### Interfaces des repositories

Les interfaces des repositories définissent les signatures des méthodes qui permettent de rechercher des entités depuis la persistance ou de persister de nouvelles entités. Lors de l’écriture de l’interface d’un repository, les points suivants sont à vérifier :

* **L’interface du repository est nommée I[Entite]Repository.**Le respect de la norme de nommage facilite la relecture du code.
* **L’interface du repository est placée dans le répertoire et l’espace de noms *Repositories*.**Le respect de cette norme améliore l’organisation du code dans les projets.
* **Les méthodes de recherche retournent des entités.**Les méthodes de recherche définies sur l’interface du repository doivent retourner des entités.
* **Les repositories des entités racines proposent une méthode pour ajouter une entité.**Les entités racines qui ne sont liées à aucune autre entité doivent être ajoutées explicitement à la persistance à l’aide d’une méthode. Le repository est l’endroit adéquat pour définir cette méthode.
* **Les méthodes qui permettent de sauvegarder les nouvelles entités sont nommées Ajouter.**Les nouvelles entités racines doivent être explicitement ajoutées sur la source de données. Le repository doit lui-même exposer une méthode Ajouter pour permettre à l’utilisateur de persister les nouvelles entités. Le respect de la norme de nommage facilite également la relecture du code.
* **Les méthodes pour obtenir une entité sont nommées Obtenir ou ObtenirDepuis[Critère].**Le respect de la norme de nommage facilite la relecture du code. S’il existe une ambiguïté entre deux méthodes de recherche car les critères de recherche sont d’un même type, on ajoute le nom du critère en suffixant la méthode par Depuis[Critère].
* **Les méthodes de recherche de listes sont nommées Lister ou ListerDepuis[Critère].**Le respect de la norme de nommage facilite la relecture du code. S’il existe une ambiguïté entre deux méthodes de recherche car les critères de recherche sont d’un même type, on ajoute le nom du critère en suffixant la méthode par Depuis[Critère].

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire l’interface d’un repository.

* [IClientRepository.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain%2FRepositories%2FIClientRepository.cs&fullScreen=true)
* [IArticleRepository.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FRepositories%2FIArticleRepository.cs&fullScreen=true)
* [ICommandeRepository.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FRepositories%2FICommandeRepository.cs&fullScreen=true)

### Implémentation des repositories

Les implémentations des repositories contiennent les requêtes qui permettent d’effectuer les recherches et les insertions définies par les interfaces. Lors de l’écriture d’un repository, les points suivants sont à vérifier :

* **Le repository est nommé [Entite]Repository.**  
  Le respect de la norme de nommage facilite la relecture du code.
* **Le repository est placé dans le répertoire et l’espace de noms *Repositories*.**Le respect de cette norme améliore l’organisation du code dans les projets.
* **Le repository possède une classe de tests unitaires dans le projet de test du domaine.**  
  Tout repository doit posséder une classe de tests unitaires. C’est un prérequis indispensable au développement selon l’approche TDD.
* **La classe de test du repository est nommée [Entite]RepositoryTest.**Le respect de la norme de nommage facilite la relecture du code et l’identification des classes de test par les outils.
* **Les tests des membres publics du repository sont écrits avant leur implémentation.**  
  Il s’agit du principe de l’approche TDD.
* **L’implémentation du repository dérive de la classe Repository.**La classe repository met à disposition une propriété de type IDataSource que l’implémentation peut utiliser pour effectuer des requêtes.
* **Le premier paramètre du constructeur est de type IDataSource.**L’implémentation du repository a besoin d’une implémentation de l’interface IDataSource pour requêter les entités. La valeur du paramètre doit être transmise au constructeur protégé de la classe de base Repository.
* **Les entités racines sont persistées à l’aide de la méthode Add<TEntity>() de IDataSource.**L’implémentation utilise la méthode Add<TEntity>() de l’objet IDataSource pour ajouter une nouvelle entité racine à la persistance.
* **Les requêtes utilisent LINQ et la méthode Query<TEntity>() de IDataSource.**LINQ permet d’écrire des requêtes typées partiellement vérifiées à la compilation. La méthode Query<TEntity>() de l’objet IDataSource permet d’obtenir un objet IQueryable<TEntity> pour écrire la requête LINQ.
* **Les requêtes LINQ sur la persistance n’utilisent que des propriétés persistées.**Les requêtes LINQ qui font références à des méthodes ou des propriétés qui ne sont pas persistées ne peuvent pas être traduites en requêtes SQL par l’ORM. Tenter d’exécuter une telle requête sur la persistance provoquera une erreur.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire l’implémentation d’un repository.

* [ArticleRepository.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FRepositories%2FArticleRepository.cs&fullScreen=true)
* [ClientRepository.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain%2FRepositories%2FClientRepository.cs&fullScreen=true)
* [CommandeRepository.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FRepositories%2FCommandeRepository.cs&fullScreen=true)

### Interfaces des systèmes externes

Les systèmes externes sont des fonctionnalités externes au domaine mais dont le domaine dépend pour fonctionner. Les interfaces des systèmes externes sont définies par le métier du domaine et reflètent les besoin du domaine vis-à-vis des éléments externes. Chaque fonctionnalité externalisée dispose de sa propre interface.

* **L’interface se trouve dans le répertoire ExternalServices.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets contenant les interfaces par les outils de configuration.

Les fichiers suivants contiennent un exemple de code qui montre comment écrire l’interface d’un système externe.

* [IReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FExternalServices%2FIReferentielClients.cs&fullScreen=true)
* [Client.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Domain%2FClient.cs&fullScreen=true)

### Interface du paramétrage métier

Le paramétrage métier est un cas particulier de service externe. Il permet d’obtenir des données paramétrées utilisées par le métier. Le métier ne connaît pas à priori la source de ce paramétrage.

* **L’interface du paramétrage est nommée IParametrage.**  
  Cette norme de nommage facilite la relecture du code.

Le fichier suivant contient un exemple de code qui montre comment écrire l’interface du paramétrage métier.

* [IParametrage.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FExternalServices%2FIParametrage.cs&fullScreen=true)

### Services métier

Les services « métier » implémentent le code de coordination entre entités qui contient de la logique métier. Ils peuvent prendre en paramètres de leurs constructeurs ou de leurs méthodes les objets qui interviennent dans la coordination : entités, registres, services externes, paramétrage, générateurs de clés… Lors de l’écriture d’un service métier, les points suivants sont à vérifier :

* **Le service métier est une classe virtuelle.**  
  Ceci est nécessaire pour créer des objets *Mock* dans les tests unitaires.
* **Le service métier possède une classe de tests unitaires dans le projet de test du domaine.**  
  Tout service métier doit posséder une classe de tests unitaires. C’est un prérequis indispensable au développement selon l’approche TDD.
* **La classe de test du service métier est nommée [ServiceMetier]Test.**Le respect de la norme de nommage facilite la relecture du code et l’identification des classes de test par les outils.
* **Les tests des membres publics du service métier sont écrits avant leur implémentation.**  
  Il s’agit du principe de l’approche TDD.
* **Tous les membres publics ou protégés du service métier sont virtuels.**  
  Ceci est indispensable pour pouvoir créer des objets *Mock* dans les tests unitaires.

Le fichier suivant contient un exemple de code qui montre comment écrire un service métier.

* [GestionnaireCommandes.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain%2FGestionnaireCommandes.cs&fullScreen=true)

## Services applicatifs

L’écriture des services applicatifs permet de sélectionner les fonctionnalités du métier que l’on souhaite exposer. Elle concerne l’écriture des interfaces des services, des implémentations des services, des objets de présentation (DTO) et des convertisseurs d’objets de présentation. En développant des services applicatifs, les points suivants sont à vérifier :

* **Les interfaces se trouvent dans le projet EIT.Fixe.[Domaine].Application.Interface.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets d’interface par les outils d’intégration continue et de configuration.
* **Les implémentations se trouvent dans le projet EIT.Fixe.[Domaine].Application.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets des services applicatifs par les outils d’intégration continue et de configuration.
* **Les tests des services se trouvent dans le projet EIT.Fixe.[Domaine].Application.Test.**Le respect de cette norme de nommage facilite la relecture du code et l’identification du projet de test par les outils d’intégration continue.

### Objets de présentation

Les objets de présentation (ou DTO) font partie des interfaces des services applicatifs. Ce sont des classes destinées à être sérialisés et véhiculées sur le réseau. Lors de l’écriture d’un objet de présentation, les points suivants sont à vérifier :

* **L’objet se trouve dans le projet EIT.Fixe.[Domaine].Application.Interface.**Le DTO est une composante de l’interface. Cette séparation est nécessaire pour le partage des assemblages d’interface. Le respect de cette norme de nommage facilite également la relecture du code.
* **L’objet se trouve dans l’espace de noms EIT.Fixe.[Domaine].Application.**Le DTO se trouve dans la couche application. Si la séparation en deux assemblages est nécessaire pour partager les interfaces, il n’est pas nécessaire de la répercuter sur l’espace de noms.
* **L’objet de présentation est une classe**.  
  Il s’agit d’une contrainte de la sérialisation.
* **L’objet de présentation est décoré avec l’attribut DataContract.**  
  Cet attribut permet d’indiquer au framework que l’objet de présentation est sérialisable.
* **L’objet de présentation possède un constructeur sans paramètre.**Le constructeur sans paramètre est nécessaire pour désérialiser l’objet de manière efficace.
* **Les propriétés sérialisables déclarent un accesseur get et un accesseur set.**Le sérialiseur doit pouvoir accéder aux propriétés sérialisables en lecture pour la sérialisation et en écriture pour la désérialisation. Les propriétés à qui il manque l’un des accesseurs ne sont pas sérialisées ou désérialisées.
* **Les propriétés sérialisables sont décorées avec l’attribut DataMember.**  
  Cet attribut permet d’indiquer au framework que la propriété de l’objet de présentation est sérialisable.
* **Les types des propriétés sérialisables font partie de la liste des types supportés.**  
  Le type des propriétés sérialisable doit faire partie de la liste des types suivants pour être supporté par le framework :
  + byte, bool, short, int, long, decimal ou DateTime
  + Un type énuméré
  + Nullable<T> ou T est l’un des types précédents
  + Un objet de présentation
  + Un tableau de l’un des types précédents

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire un objet de présentation.

* [Client.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Interface%2FClient.cs&fullScreen=true)
* [Article.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application.Interface%2FArticle.cs&fullScreen=true)
* [Commande.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application.Interface%2FCommande.cs&fullScreen=true)

### Interfaces des services

Les interfaces des services définissent les fonctionnalités exposées de l’application sous forme de méthodes. Lors de l’écriture de l’interface d’un service, les points suivants sont à vérifier :

* **L’interface se trouve dans le projet EIT.Fixe.[Domaine].Application.Interface.**Cette norme de nommage facilite la relecture du code et l’identification des projets interface par les outils d’intégration continue et de configuration. Cette séparation est nécessaire pour le partage des assemblages d’interface.
* **L’interface se trouve dans l’espace de noms EIT.Fixe.[Domaine].Application.**L’interface du service applicatif se trouve dans la couche application. Si la séparation en deux assemblages est nécessaire pour partager les interfaces, il n’est pas nécessaire de la répercuter sur l’espace de noms.**L’interface d’un service synchrone est nommée I[Nom]Service.**  
  Cette norme de nommage facilite la relecture du code.
* **L’interface d’un service de consommation de messages est nommée I[Nom]EventService.**Cette norme de nommage facilite la relecture du code.
* **Les services synchrones et de consommation des messages ont des interfaces séparées.**Cette norme facilite la relecture du code.
* **L’interface est décorée avec l’attribut ServiceContract.**Cet attribut permet d’indiquer au framework que l’interface définit un service à exposer.
* **Les méthodes de l’interface sont décorées avec l’attribut OperationContract.**Cet attribut permet d’indique au framework que la méthode est une opération qui doit être exposée.
* **Le type de retour des méthodes fait partie de la liste des types supportés.**Les méthodes définies dans les interfaces des services doivent retourner des objets de présentation ou l’un des types supportés par les objets de présentation. Les services applicatifs ne doivent pas retourner de types du code métier.
* **Les méthodes de consommation des événements ont le type de retour void.**La consommation des événements n’a aucun retour. Ce retour ne serait pas exploitable ni par l’émetteur du message ni par le broker de messages.
* **Les types des paramètres des méthodes font partie de la liste des types supportés.**Les paramètres des méthodes définies dans les interfaces des services doivent correspondre à des objets de présentation ou à l’un des types supportés par les objets de présentation. Les services applicatifs ne doivent pas demander des types du code métier.
* **Si elle est requise par le service, l’identité est le premier paramètre de toutes les méthodes.**

Placer l’identité du paquet EIT.Fixe.Systeme.Identification en paramètre des méthodes permet de journaliser automatiquement les informations d’identification fournies par l’appelant. Le placement systématique du paramètre de l’identité en première position facilite la relecture du code.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire l’interface d’un service applicatif.

* [IArticleService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application.Interface%2FIArticleService.cs&fullScreen=true)
* [ICommandeService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application.Interface%2FICommandeService.cs&fullScreen=true)
* [IClientService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Interface%2FIClientService.cs&fullScreen=true)
* [IClientEventService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Interface%2FIClientEventService.cs&fullScreen=true)

### Implémentation des services

Les implémentations des services utilisent les repositories pour rechercher des entités et appeler les bonnes méthodes sur ces dernières. Lors de l’écriture de l’implémentation d’un service, les points suivants sont à vérifier :

* **L’implémentation se trouve dans le projet EIT.Fixe.[Domaine].Application.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets des services par les outils d’intégration continue et de configuration.
* **L’implémentation d’un service synchrone est nommée [Nom]Service.**  
  Le respect de la norme de nommage facilite la relecture du code.
* **L’implémentation d’un service de consommation de messages est nommée [Nom]EventService.**  
  Le respect de la norme de nommage facilite la relecture du code.
* **Les services synchrones et de consommation des messages ont des implémentations séparées.**Cette norme facilite la relecture du code.
* **Le service possède une classe de tests unitaires dans le projet de test des services.**  
  Tout service doit posséder une classe de tests unitaires. C’est un prérequis indispensable au développement selon l’approche TDD.
* **La classe de test du service est nommée [Nom]ServiceTest.**Le respect de la norme de nommage facilite la relecture du code et l’identification des classes de test par les outils.
* **Les tests des méthodes du service sont écrits avant leur implémentation.**  
  Il s’agit du principe de l’approche TDD.
* **Le constructeur demande en paramètres les interfaces des objets dont le service a besoin.**Les services applicatifs ont besoin de travailler avec des modules métiers ou techniques définis par des interfaces mais dont l’implémentation n’est à priori pas connue. Ils doivent récupérer ces objets depuis les arguments de leur constructeur. Les implémentations seront fournies par l’injection de dépendances du framework. Les modules peuvent comporter :
  + Les repositories
  + Les systèmes externes
  + Les générateurs de clé
  + Les fournisseurs de paramétrage
* **Le service conserve dans des champs les implémentations des objets dont il a besoin.**Les implémentations récupérées depuis les arguments du constructeur doivent être conservés dans des champs privés pour pouvoir être réutilisés dans les méthodes.
* **Les clés des entités sont générées avec le générateur de clés.**Lorsqu’un service applicatif doit créer des entités il doit générer la clé avec le générateur de clé du paquet EIT.Fixe.Systeme.Persistance. Le générateur de clé permet d’obtenir une clé entière sans introduire de couplage supplémentaire à la base de données.
* **Les services applicatifs ne doivent pas contenir de logique métier.**La logique métier doit être entièrement contenue dans le projet EIT.Fixe.[Domaine].Domain de telle sorte qu’on puisse changer la couche applicative sans perdre d’information sur le fonctionnement du métier. Le rôle du service doit se limiter à la rechercher dans les registres et à quelques appels de méthodes. S’il faut matérialiser des règles complexes de coordination entre entités, il faut créer un service « métier ».
* **Le code de conversion des objets de présentation est factorisé.**Les méthodes des services applicatifs sont amenées à convertir des objets de présentations vers ou depuis des entités du métier et inversement. Le code qui effectue ces conversions doit être factorisé dans des convertisseurs pour faciliter la maintenance du code.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire l’implémentation d’un service applicatif.

* [ClientService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientService.cs&fullScreen=true)
* [ClientEventService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientEventService.cs&fullScreen=true)
* [CommandeService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application%2FCommandeService.cs&fullScreen=true)
* [ArticleService.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application%2FArticleService.cs&fullScreen=true)

### Convertisseurs

Les classes de conversions sont des outils qui permettent de convertir les entités en objets de présentations. Lors de l’écriture de méthodes de conversion, les points suivants sont à vérifier :

* **Les méthodes de conversion se trouvent dans le projet EIT.Fixe.[Domaine].Application.**Les classes de conversion manipulent des objets métier et ne doivent pas faire partie des interfaces.
* **Les méthodes de conversion groupées par entité sur des classes statiques.**  
  Les méthodes de conversions sont partagées entre tous les services applicatifs du domaine sans surcoût en termes de code. Le regroupement par entité facilite la relecture du code.
* **La classe de conversion est nommée [NomEntite]Mapper.**Cette norme de nommage facilite la relecture du code.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire un convertisseur.

* [ArticleMapper.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application%2FArticleMapper.cs&fullScreen=true)
* [CommandeMapper.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Application%2FCommandeMapper.cs&fullScreen=true)
* [ClientMapper.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application%2FClientMapper.cs&fullScreen=true)

## Tests unitaires

Les tests unitaires sont présents à différentes étapes du développement. En écrivant des tests, les points suivant sont à vérifier :

* **Les tests sont contenus dans un projet EIT.Fixe.\*.Test.**Le respect de cette norme de nommage facilite la relecture du code et l’identification des projets de test par les outils d’intégration continue.
* **Les méthodes de test sont décorées avec l’attribut Test.**L’attribut Test indique à NUnit que la méthode est un test unitaire.
* **Les méthodes de test sont écrites selon les trois sections *Arrange*, *Act* et *Asset*.**Le respect de cette norme facilite la relecture du code. La première section *Arrange* correspond à la préparation des données, la deuxième section *Act* à l’appel de la méthode, de la propriété ou du constructeur et la troisième section *Assert* à la vérification du résultat.
* **Les méthodes de test sont nommées [Action]\_[Condition]\_[ResultatAttendu].**Afin de faciliter la relecture du code, les méthodes de test ont un nom explicite qui facilite la relecture du code. La première partie du nom décrit l’action, la deuxième la condition de l’action et la troisième le résultat attendu.
* **Le générateur de clé est *mocké* avec la classe EIT.Fixe.Systeme.Tests.GenerateurCles.**Le paquet EIT.Fixe.Systeme.Tests fournit une implémentation en mémoire du générateur de clé. C’est la solution de *mock* unifiée pour le générateur de clés.
* **Les registres utilisent la source de données EIT.Fixe.Systeme.Tests.DataSource.**Le paquet EIT.Fixe.Systeme.Tests fournit une implémentation en mémoire de la source de données. C’est la solution de *mock* unifiée pour la source de données utilisée par les tests des registres.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire des tests unitaires.

* [ClientServiceTest.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Clientele.Application.Test%2FClientServiceTest.cs&fullScreen=true)
* [CommandeRepositoryTest.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain.Test%2FRepositories%2FCommandeRepositoryTest.cs&fullScreen=true)
* [ArticleTest.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Domain.Test%2FArticleTest.cs&fullScreen=true)

## Services externes

Les services externes requis par le domaine et définis sous forme d’interfaces dans le métier sont implémentés dans un projet technique séparé afin que le métier reste indépendant. Le métier ne doit jamais dépendre d’une implémentation précise d’un service externe. Lors de l’écriture d’un service externe, les points suivants sont à vérifier :

* **L’implémentation se trouve dans le projet EIT.Fixe.[Domaine].Infrastructure.ExternalServices.**  
  Le respect de cette norme facilite la relecture du code et l’identification des projets contenant les implémentations par les outils de configuration.

### Interface du paramétrage technique

Le paramétrage technique est le paramétrage nécessaire au fonctionnement des systèmes externes. Il permet de configurer l’interconnexion technique avec les éléments externes. Son interface ne concerne par le métier et est directement définie par les services externes.

* **L’interface du paramétrage est nommée IParametrage[NomServiceExterne].**  
  Cette norme de nommage facilite la relecture du code.

Le fichier suivant contient un exemple de code qui montre comment écrire l’interface du paramétrage technique.

* [IParametrageReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FIParametrageReferentielClients.cs&fullScreen=true)

### Implémentation du paramétrage

Le paramétrage métier est un cas particulier de service externe. Son implémentation est standardisée et utilise la base de données Oracle. Lors de l’écriture d’un service externe de paramétrage, les points suivants sont à vérifier :

* **L’implémentation est nommée Parametrage.**  
  Le respect de la norme de nommage facilite la relecture du code.
* **L’implémentation dérive de la classe TableParametrage.**La classe TableParametrage est la solution standardisée du projet *EIT.Fixe* pour récupérer le paramétrage dans une table de la base de données Oracle et gérer la mise en cache des paramètres.
* **L’implémentation utilise le fournisseur de connexion FournisseurConnexion.**La classe FournisseurConnexion centralise la configuration de la connexion à la base de données.
* **L’implémentation passe le nom de la table de paramétrage à la classe de base.**En dérivant de la classe FournisseurConnexion, l’implémentation du paramétrage doit donner le nom de la table de paramétrage au constructeur protégé de la classe mère.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire l’implémentation du paramétrage avec une table Oracle.

* [Parametrage.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FParametrage.cs&fullScreen=true)
* [ParametrageReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FParametrageReferentielClients.cs&fullScreen=true)

### Service applicatif d’un autre domaine interne

Les appels à un service applicatif d’un autre domaine interne est un cas particulier de service externe. Lors de l’écriture d’un tel service externe, les points suivants sont à vérifier :

* **Le constructeur prend l’interface du service de l’autre domaine en paramètre.**  
  Pour obtenir une implémentation du service d’un autre domaine, l’implémentation du service externe demande l’interface correspondante dans les arguments de son constructeur. L’injection de dépendances se chargera de fournir la bonne implémentation.

Le fichier suivant contient un exemple de code qui montre comment écrire l’implémentation d’un service externe pour l’appel d’un autre domaine interne.

* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

### Service applicatif synchrone d’un autre domaine externe

Les appels à un service applicatif d’un domaine externe à l’application est un cas particulier de service externe de type WCF. Lors de l’écriture d’un tel service externe, les points suivants sont à vérifier :

* **L’implémentation utilise l’interface IClientFactory pour instancier les clients WCF.**L’interface IClientFactory est la solution centralisée et standardisée du projet *EIT.Fixe* pour créer des clients WCF.
* **Le constructeur prend un argument de type IClientFactory.**Pour obtenir une implémentation de la fabrique de client, l’implémentation du service externe demande l’interface correspondante dans les arguments de son constructeur. L’injection de dépendances se chargera de fournir la bonne implémentation.
* **Le constructeur prend un argument de type IParametrage[NomServiceExterne].**Le service externe a besoin du paramétrage pour obtenir la configuration des services distants, telle que l’URL. Pour obtenir une implémentation du paramétrage, l’implémentation du service externe demande l’interface correspondante dans les arguments de son constructeur. L’injection de dépendances se chargera de fournir la bonne implémentation.
* **Les données techniques sont correctement paramétrées.**  
  L’URL, les informations de connexion et les diverses données techniques qui permettent d’établir la communication avec le système externe ne doivent pas être en dur dans le code de l’implémentation mais utiliser le paramétrage technique.
* **Les instances des clients WCF sont utilisées avec le mot clé using.**Les objets IClients retournés par l’interface IClientFactory implémentent IDisposable. L’usage du mot clé using garantit que les instances des clients sont correctement disposées.

Le fichier suivant contient un exemple de code qui montre comment écrire l’implémentation d’un service externe pour l’appel d’un domaine externe avec WCF.

* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

### Communication asynchrone par commandes avec un autre domaine

L’envoi de messages asynchrones de type *commande* vers un autre domaine est un cas particulier de service externe. Lors de l’écriture d’un tel service externe, les points suivants sont à vérifier :

* **L’implémentation utilise l’interface IMessagingSystem pour envoyer les messages.**L’interface IMEssagingSysyem est la solution fournie par le framework EIT pour envoyer des messages. Dans le cas de messages de type *commande*, les messages sont directement envoyés par cette interface depuis un service externe.
* **Le constructeur prend un argument de type IMessagingSystem.**Pour obtenir une implémentation du système d’envoi de messages, l’implémentation du service externe demande l’interface correspondante dans les arguments de son constructeur. L’injection de dépendances se chargera de fournir la bonne implémentation.

Le fichier suivant contient un exemple de code qui montre comment écrire l’implémentation d’un service externe qui envoie des messages de type *commande*.

* [ReferentielClients.cs](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.ExternalServices%2FReferentielClients.cs&fullScreen=true)

### Web Service avec une Web Reference

Dans le cas où le domaine externe ou le partenaire distant ne propose pas d’interfaces WCF, le service externe devra générer une Web Reference à partir d’un fichier WSDL. Lors de l’écriture d’un tel service externe, les points suivants sont à vérifier :

* **Le constructeur prend un argument de type IParametrage[NomServiceExterne].**Le service externe a besoin du paramétrage pour obtenir la configuration du Web Service, telle que l’URL. Pour obtenir une implémentation du paramétrage, l’implémentation du service externe demande l’interface correspondante dans les arguments de son constructeur. L’injection de dépendances se chargera de fournir la bonne implémentation.
* **Les données techniques sont correctement paramétrées.**  
  L’URL, les informations de connexion et les diverses données techniques qui permettent d’établir la communication avec le système externe ne doivent pas être en dur dans le code de l’implémentation mais utiliser le paramétrage technique.

## Scripts SQL

Les entités du code métier sont mappées sur des tables Oracle. En écrivant les scripts SQL de création des tables et des séquences et indexes associés, les points suivants sont à vérifier :

* **Le tablespace des objets est toujours explicitement renseigné.**Si le tablespace n’est pas spécifié, l’objet sera créé dans le tablespace par défaut. Placer les données dans le même tablespace par défaut empêche la gestion fine de la taille de la base.
* **L’objet est placé dans le tablespace attribué à son domaine.**Les données sont réparties dans différents tablespaces en fonction du domaine auquel elles appartiennent. Ceci permet de gérer la taille de la base domaine par domaine. Les tablespaces sont les suivants :
  + TBS\_EIFF pour la facturation
  + TBS\_EIFO pour les offres
  + TBS\_EIFP pour la vie client et le parc
  + TBS\_EIFS pour la souscription
* **La table est nommée « T\_[NomTable] ».**Cette norme de nommage est requise pour l’utilisation module de génération de clés. Elle facilite l’identification des objets dans la base de données.
* **Si la table est une table de paramètres, elle est nommée « T\_[Préfixes]\_PARAMETRE ».**Cette norme de nommage facilite la relecture du code et l’identification des tables de paramètres.
* **La séquence de la clé de la table est nommée « S\_[NomTable] ».**Cette norme de nommage est requise pour l’utilisation du module de génération de clés. Elle facilite l’identification des objets dans la base de données.
* **La valeur maximale des séquences correspond à la taille de la colonne clé correspondante.**Ceci permet de lever une erreur dans le cas où l’on arriverait à la valeur maximale.
* **Les tables des entités dérivées n’ont pas de séquence associée.**La clé présente dans la table d’une entité fille est la même que celle de la table de l’entité mère. Il n’y a donc pas besoin de séquence pour les tables des entités dérivées.
* **Le nom des indexes est préfixé par IDX\_ .**Cette norme de nommage facilite l’identification des objets dans la base de données.
* **Les contraintes sont toujours nommées.**Cette norme facilite la comparaison entre deux bases. En effet, si les contraintes ne sont pas nommées, leur nom est automatiquement généré par la base et peut être différent entre deux bases.
* **Le nom des contraintes de clés primaires débute par PK\_.**Le respect de cette norme facilite la relecture du code en indiquant dans le nom la nature de la contrainte.
* **Le nom des contraintes de clés étrangères débute par FK\_.**Le respect de cette norme facilite la relecture du code en indiquant dans le nom la nature de la contrainte.
* **Le nom des contraintes d’unicité débute par UK\_.**Le respect de cette norme facilite la relecture du code en indiquant dans le nom la nature de la contrainte.
* **Le nom des contraintes de non-nullité débute par CHK\_.**Le respect de cette norme facilite la relecture du code en indiquant dans le nom la nature de la contrainte.
* **Le nom des colonnes des clés étrangères qui référencent une clé est préfixé par CLE.**Le respect de cette norme facilite la relecture du code en indiquant dans le nom qu’il s’agit d’une clé étrangère.
* **Les abréviations des noms sont des trigrammes provenant de la *liste des abréviations*.**Oracle limitant le nom des symboles à 30 caractères, il est parfois nécessaire d’utiliser des abréviations dans le nom des tables, des colonnes, des indexes et des séquences. Afin de standardiser le nommage, les abréviations utilisées sont les trigrammes disponibles dans la *liste des abréviations* (LISABR dans la zone express).
* **Le type des colonnes Oracle correspond au type CLR des propriétés des entités.**Afin de garantir le bon fonctionnement de l’ORM, le type des colonnes de la base est imposé par le type des propriétés des entités. Le type Les correspondances entre les types sont les suivantes :
  + bool correspond à NUMBER(1)
  + DateTime correspond à TIMESTAMP(7)
  + decimal correspond à NUMBER(X, Y)   
    X est le nombre total de chiffres  
    Y est le nombre de chiffres après la virgule
  + short correspond à NUMBER(4)
  + int correspond à NUMBER(9)
  + long correspond à NUMBER(18)
  + string correspond à VARCHAR(X)  
    X est le nombre de caractères maximal de la chaîne, il ne peut pas dépasser 2000
  + Pour les énumérations, on utilise le type correspondant au type CLR sous-jacent  
    Par défaut le type sous-jacent est int donc le type correspondant est NUMBER(9)
* **Toute colonne qui correspond à une clé étrangère est indexée.**Lors de la suppression d’une ligne ou de la modification d’une clé dans une table, Oracle va vérifier l’intégrité des données en parcourant toutes les clés étrangères qui référencent la table. Pour éviter les problèmes de performance, on impose la création d’un indexe sur chaque clé étrangère.
* **Une collection persistante est toujours matérialisée par une table d’association.**Toutes les collections persistantes que l’on retrouve dans le code métier sont matérialisées par une table d’association dans la base Oracle. Ceci permet à une classe parente de référencer des tables enfant sans imposer de clés étrangères dans les tables enfant.
* **La colonne de l’état d’une entité est nommée ETAT\_VALEUR.**Ce nom est nécessaire pour le bon fonctionnement de la classe de gestion de la machine à états d’une entité.

Les fichiers suivants contiennent des exemples de code qui montrent comment écrire le script d’une table d’entité et une table d’association.

* [T\_EPR\_CDE\_COMMANDE.sql](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.Repository.Oracle%2FT_EPR_CDE_COMMANDE.sql&fullScreen=true)
* [T\_EPR\_CDE\_COMMANDE\_LIGNECDE.sql](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.Repository.Oracle%2FT_EPR_CDE_COMMANDE_LIGNECDE.sql&fullScreen=true)

### Tables de paramètres

Le module de paramétrage EIT.Fixe.Systeme.Parametrage.Oracle s’appuie sur une structure précise des tables de paramètres dans la base Oracle. En écrivant une table de paramétrage, les points suivants sont à vérifier :

* **La table possède une colonne CLE et une colonne VALEUR toutes deux de type VARCHAR2(X).**Cette norme de nommage est requise pour l’utilisation du module de paramétrage.
* **La clé primaire est la colonne CLE.**Cette norme est requise pour l’utilisation du module de paramétrage.
* **La colonne VALEUR possède une contrainte NOT NULL.**Cette norme permet d’éviter les erreurs à l’utilisation du module de paramétrage.
* **La table est nommée T\_[TrigrammeDomaine]\_PARAMETRE.**Cette norme de nommage facilite la relecture du code en indiquant la nature de la table dans la base de données.

Le fichier suivant contient un exemple de code qui montre comment écrire le code d’une table de paramètres.

* [T\_EPR\_CDE\_PARAMETRE.sql](http://tfs.eit.cm-cic.fr:8080/tfs/EITCollection/Fixe/_git/EIT.Fixe.Exemple?path=%2FEIT.Fixe.Exemple.Commandes.Infrastructure.Repository.Oracle%2FT_EPR_CDE_PARAMETRE.sql&fullScreen=true)

# Questions et problèmes fréquents

Cette section regroupe les questions fréquemment posées et les problèmes fréquemment rencontrés lors des développements avec le framework EIT et les outils spécifiques aux développements du projet *EIT.Fixe*.