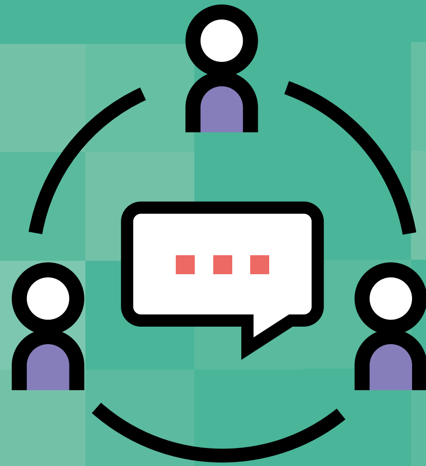
SOAT

→ Digitalize society



Domain Driven Design

Avril 2021

Romain Deneau – Sepehr Namdar Fard

SOAT.FR

Table des matières

→ Introduction 03

→ Collaborative Modeling 11

→ Strategic Design 30

→ Tactical Design 45

→ Conclusion 60

→ Retours d’expérience 62

Remerciements

La rédaction de ce livre blanc a demandé un travail collectif sur une durée d'un an.

En plus des rédacteurs et relecteurs, ce livre blanc n'aurait pas pu aboutir sans la présence de certaines personnes clés.

Nous tenons à remercier notre CTO Guillaume NURDIN, qui a créé un environnement nous permettant de fournir un travail de qualité.

Ania EMAMI, qui a assuré la qualité de la rédaction de ce livre blanc.

Gregory BOISSINOT, ancien CTO de SOAT, qui nous a transmis son savoir-faire et son expertise en particulier sur DDD.

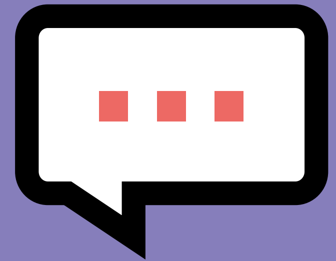
En espérant que le travail fourni soit à la hauteur de leur investissement.

Introduction

Objectif du livre

L'objectif de ce livre blanc est d'offrir une porte d'entrée au DDD, couvrant ses différents aspects jusqu'aux plus récents, le tout sans se perdre dans des subtilités philosophiques ou techniques.

La notion de DDD provient du livre *Domain-Driven Design: Tackling Complexity in the Heart of Software* d'Eric Evans sorti en 2003. C'est encore aujourd'hui la référence, appelé *Big Blue Book*, pour saisir l'essence et la portée du DDD. Son inconvénient est d'être volumineux, quasiment 600 pages, et difficile à lire, d'autant plus qu'il n'existe qu'en anglais. D'autres livres écrits par des auteurs tels que Vaughn Vernon, Nick Tune et Scott Millet ont suivi une dizaine d'années plus tard, c'est dire le temps de maturation nécessaire lorsque l'on parle de DDD. Ces livres sont plus approfondis, en particulier sur le plan technique. On reste toutefois face à de «gros pavés» de 500 à 1000 pages. Il existe aussi quelques «abrégés», mais aucun rédigé en français que nous trouvons satisfaisants.



DDD en trois mots

Le DDD est un sujet vaste et complexe, le définir est une tâche ardue. Cela pourrait expliquer pourquoi, dans l'industrie du logiciel, il n'est pas bien compris et encore moins pratiqué. Voici quelques éléments pour y voir plus clair.

Commençons par une définition très proche de celle de *Scott Millet* :

> DDD est une approche de conception de logiciel. Cette approche est centrée sur le métier, pour des domaines présentant des problématiques complexes.
>
> DDD regroupe un ensemble de principes, de pratiques et de patterns qui vont permettre aux équipes de se concentrer sur ce qui est essentiel à la réussite commerciale, tout en façonnant un logiciel à même de gérer la complexité à la fois technique et métier.

Cette définition est rigoureuse, d'où sa longueur, qui en fait aussi un inconvénient. Essayons de simplifier sans perdre en substance. Si l'on devait s'en tenir à quelques mots pour désigner le DDD, les trois termes suivants conviendraient bien car ils condensent le titre complet du *Big Blue Book* :

Métier · Complexité · Modèle (MCM)

Métier

Le *Métier* est synonyme de *Domaine* tout en étant plus simple à appréhender. Le *Métier* est une préoccupation permanente et essentielle dans le DDD, la raison d'être des logiciels de notre industrie étant de répondre à un besoin métier.

> *« We **[as developers]** are not **code writers**, we are **problem solvers**. Sometimes the best approach is not to write code or **[is]** to get rid of useless code. »*

Michael Feather[1]

Complexité

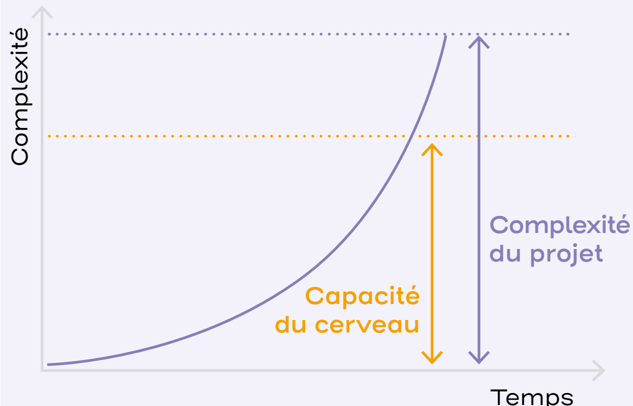
La *Complexité* est le nerf de la guerre du DDD. Elle désigne à la fois la complexité métier et la complexité technique. L'objectif est de s'attaquer à la complexité afin de la réduire. Le DDD s'adresse à un domaine métier complexe, avec une « sauce maison » qui en fait un facteur de différentiation de l'entreprise par rapport à son éventuelle concurrence.

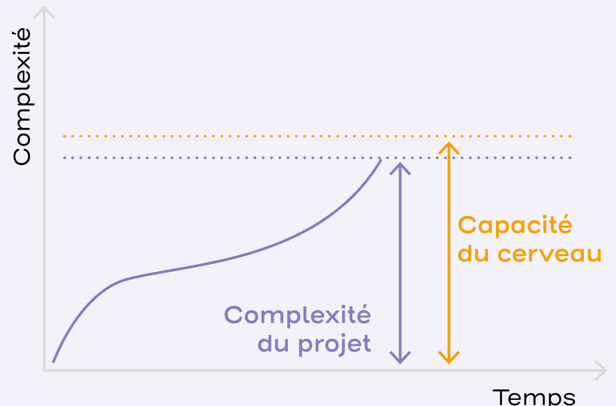
Modèle

Le dernier terme, *Modèle*, en découle d'un autre, *Design*, le troisième D de DDD. Il est plus difficile à saisir. Il faut se rappeler que tout logiciel est façonné au moyen d'une modélisation du *Problème* et a pour but d'en apporter une *Solution*. Le modèle est donc la clé de voûte du logiciel.

1. Auteur du livre *Working Effectively With Legacy Code*

La complexité dans la réalisation de logiciels





L'objectif du DDD est de maintenir la complexité en dessous d'un certain seuil, compréhensible par le cerveau humain. Cela permet de conserver un produit maintenable dans le temps.

La complexité présente deux composantes.

- La complexité inhérente au métier
  - *Essentielle*
- *La complexité technique et du Legacy*
  - *Accidentelle*

Voyons différentes problématiques qui rendent complexe la mise au point d'un logiciel évolutif.

Métier relégué au second plan

L'image ci-contre d'un projet Java est symptomatique du passage au second plan du métier *(Entretien)* au profit de la technique *(Controller, DAO, DTO, Service)*.

```
src
  controller
    EntretienController.java
  dao
    EntretienDao.java
  dto
    EntretienDto.java
  service
    EntretienService.java
```

On parle d'architecture aveugle. En guise de remède, Oncle Bob prône la *Screaming Architecture*[2], une architecture logicielle où le métier saute aux yeux.

Mais il ne s'agit ici que d'un exemple. Le problème est plus profond, à tout niveau dans le code et surtout dans les mentalités.

Le code est un outil qui nous permet de mettre en place une solution.

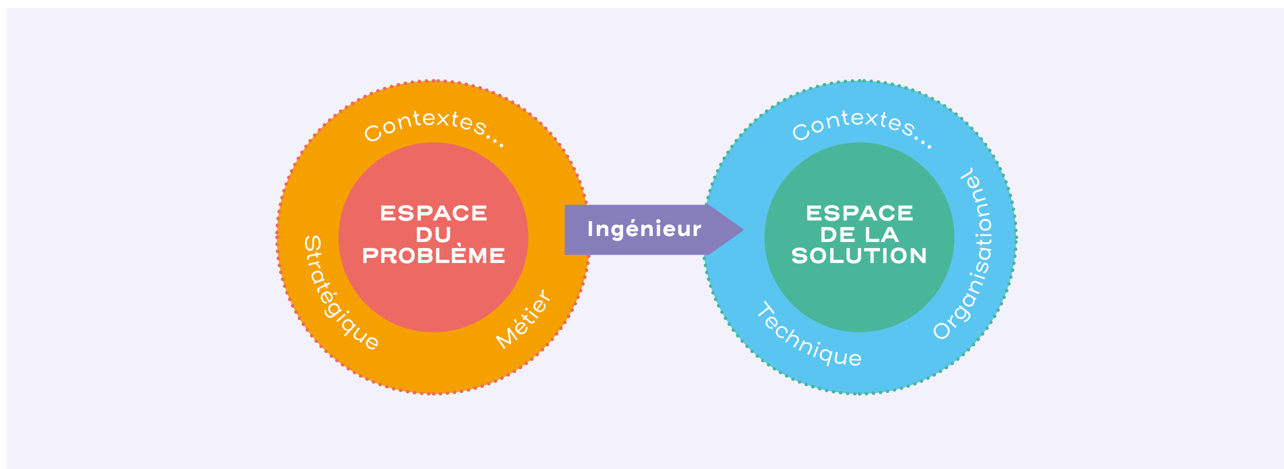
2. https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html, Robert C. Martin, 2011

*«**Software** is **design**. **Code** is just the tool to **implement** the design.»*

Mathias Verraes[3]

Ingénierie logicielle

Le métier de développeur devrait être un vrai métier d'ingénierie. L'ingénieur est celui qui fait passer la réalité de l'espace du *Problème* à celui de la *Solution* en faisant preuve de pragmatisme.



Cela implique deux choses :

1. Le problème métier doit être la préoccupation première des acteurs. **Il faut comprendre ce métier, parler métier, raisonner métier.** Toute décision doit être prise dans le contexte le plus large possible. Alors, peut se mettre en place une véritable collaboration avec les acteurs métier, à la recherche de la solution, ce qui peut éventuellement se concrétiser dans la réalisation d'un produit.
   - → *Les développeurs qui ne pensent que technique se fourvoient.*
   - → *Les développeurs devraient être évalués non pas sur la rapidité à produire du code mais sur la façon de comprendre le métier, puis de résoudre des problèmes, avec ou sans code.*

2. L'ingénieur est un être de terrain qui trouve une solution adaptée et généralement spécifique, chaque entreprise voulant trouver sa propre «sauce secrète».
   - → *Il s'agit rarement d'appliquer des recettes génériques comme certains progiciels aimeraient le faire croire.*

*«**Critical complexity** of most software projects is in understanding the **domain** itself.»*

Eric Evans

3. Co-organisateur de DDD Europe, Software Modelling & Design Conference

L'IT participe au business de l'entreprise, de manière encore plus stratégique qu'auparavant dans bon nombre de métiers. Il faut chercher à ce que cette vision soit partagée par toutes les parties prenantes, **l'accord au sein du groupe est donc primordial.** Il ne s'agit pas de rejeter en bloc une ébauche de solution venant des experts métier, mais plutôt de rappeler que chacun est expert dans son domaine (métier, organisationnel, technique...).

Les acteurs du métier ne sont pas experts en tout et ne peuvent pas avoir réponse à tout. De même, ce n'est pas aux équipes techniques d'imposer des solutions impactant le métier ou le process, mais chacun peut apporter des choses à l'autre. C'est un terrain délicat, «l'humain».

Un respect mutuel est synonyme d'une meilleure collaboration entre tous.

Absence de langage commun

Mettre le métier au premier plan ne suffit pas. Il faut également trouver un moyen de communication efficace. Souvent, l'équipe de développement et le métier ne parlent pas le même langage.

Voici quelques exemples :

	ÉQUIPE DE DÉVELOPPEMENT	MÉTIER
Terme inconnu du métier	Parle de *Spécification Entretien*.	Cela ne parle pas au métier.
Plusieurs termes, un seul concept	Utilise le terme *Client*.	Le terme métier est *Contact*.
Terme avec différents concepts	Une *Salle* a juste un *Nom*.	Une *Salle* est également reliée à un *Étage*, comporte un certain *Nombre de places* et peut disposer de certains *Équipements*.
Langues différentes	Utilise l'anglais : *Vehicle, Amendment...*	Parle en français : *Voiture, Avenant...*

Ces problèmes sont d'autant plus insidieux qu'ils ne sont pas toujours manifestes. Il arrive fréquemment que certaines choses semblent évidentes voire instinctives pour les experts métier, telles que des subtilités dans les concepts et dans les processus, mais qu'elles ne soient pas connues de l'équipe de développement, ce qui implique qu'elles ne pourront pas être intégrées dans les systèmes. Ces systèmes et ceux qui s'en servent ne sont pas alignés, et cela crée de la friction.

Les mêmes travers peuvent également se retrouver au sein même de l'équipe de développement. Il existe souvent une disparité de connaissance métier entre les différents membres d'une équipe. Lorsque l'on explique quelque chose qui n'est pas parfaitement clair pour nous, cela sera encore plus flou pour les autres sans un effort collectif de clarification.

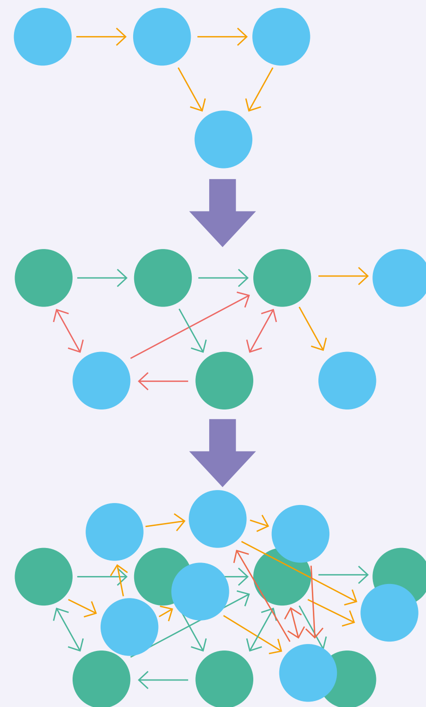
Les exemples que nous venons de citer sont des symptômes de défauts de communication. Ils peuvent être résolus en favorisant la collaboration entre les participants du projet, tout en restant **rigoureux** et **vigilants**.

> *« If you can't explain it* ***simply****, you don't* ***understand*** *it well enough. »*
>
> Albert Einstein

Manque de soins

Le métier de développeur devrait être un vrai métier d'ingénierie. L'ingénieur est celui qui fait passer la réalité de l'espace du *Problème* à celui de la *Solution* en faisant preuve de pragmatisme.

1. *Les premières versions d'un logiciel sont rapides à produire, d'autant plus si l'on néglige certains aspects au niveau qualité. Des petits défauts de conception peuvent être déjà perceptibles. Ils ne coûtent alors pas cher à corriger. Cette rapidité de production peut être grisante et peut amener à ignorer certains défauts.*
2. *Au fur et à mesure des versions, la mise de côté des soins se manifeste par une augmentation exponentielle de la complexité. Les développeurs passent de plus en plus de temps aux prises avec de la complexité technique plutôt qu'à traiter celle du métier. L'image est celle du plat de spaghetti.*
3. *La dégradation se poursuivant, la charge mentale pour comprendre le code et le modifier atteint les limites des cerveaux mêmes les plus brillants. Le logiciel continue de fonctionner, mais plus personne ne comprend comment et n'ose se risquer à le modifier. L'image est alors celle d'une* ***Big Ball of Mud****.*

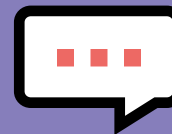


Ce phénomène a été nommé *Dette Technique*[4] par analogie avec une dette financière : l'effort supplémentaire requis pour ajouter de nouvelles fonctionnalités correspond à l'intérêt généré par la dette. Le stade 3 ci-dessus peut rapidement dégénérer en banqueroute.

4. Depuis, l'expression *Dette technique* est plutôt utilisée pour indiquer des choix de raccourcis dans le design tout en conservant un code de qualité. Un code de mauvaise qualité n'est plus considéré comme de la dette technique aujourd'hui mais du *Legacy* problématique. https://fr.wikipedia.org/wiki/Dette_technique

À tout niveau dans un système, du code le plus profond aux architectures des logiciels et du SI, des soins réguliers sont nécessaires pour garder les logiciels maintenables.

La maintenabilité[5] est la valeur essentielle dans la conception d'un logiciel, avant même son bon fonctionnement, pour rester le plus aligné possible avec la vision métier au gré de ses évolutions.



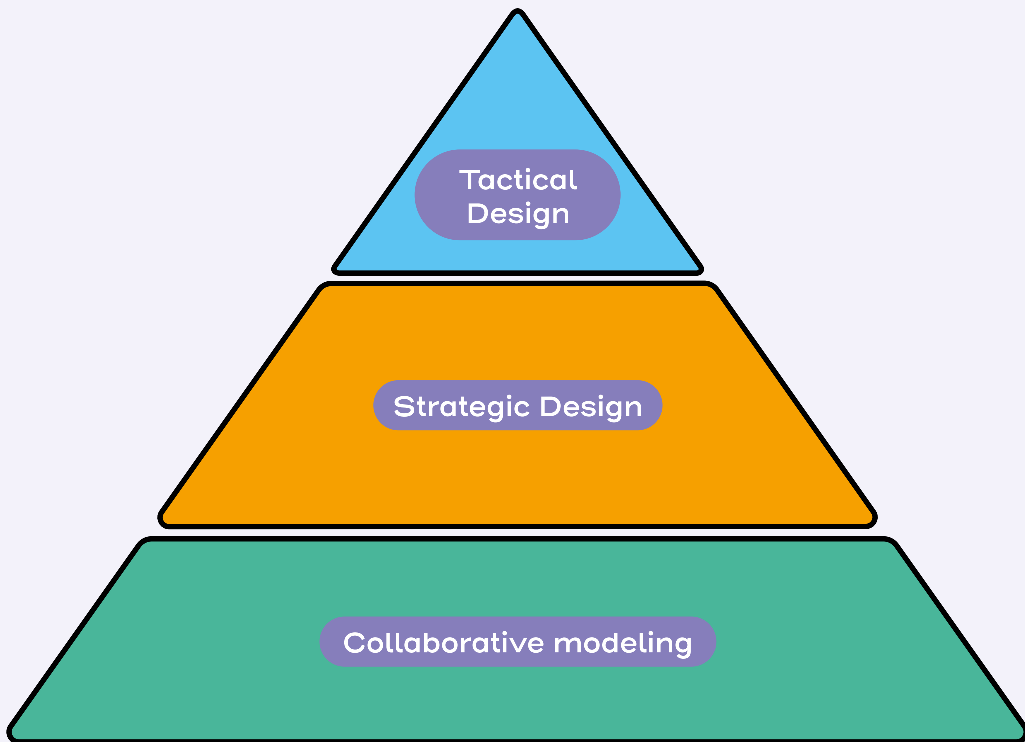
Les trois piliers du DDD

Le DDD s'articule autour de trois piliers, des thématiques complémentaires faisant chacune l'objet d'un chapitre dédié, dont voici un aperçu pour s'en faire une idée générale :

- **Collaborative modeling :** permet de remettre le métier en tant que préoccupation principale. La modélisation permet de passer de l'espace du problème à l'espace de la solution. Elle se fait de manière collaborative entre l'équipe de développement, les experts métier et les autres acteurs du projet.
- **Strategic design :** le mot d'ordre est de diviser pour mieux garder le contrôle. Il s'agit de transformer le modèle de domaine brut pour qu'il soit exploitable et traduisible dans un logiciel, en le partitionnant pour en diminuer la complexité. Il s'agit aussi de situer le produit dans son écosystème, de délimiter ses frontières et de qualifier leurs relations.
- **Tactical design :** c'est la partie qui parle le mieux aux aficionados de la technique mais la moins importante dans le DDD. On y trouve des building blocks dont l'usage éclairé permet d'améliorer le design du code et de réduire la complexité accidentelle.

Le meilleur du DDD s'obtient en faisant de chaque pilier la fondation du suivant, dans l'ordre indiqué : le *collaborative modeling* est le plus fondamental et permet de réaliser un meilleur *strategic design*, qui lui-même soutient le *tactical design*.

5. Dans le sens de l'adaptation au changement pour livrer fréquemment un logiciel qui fonctionne

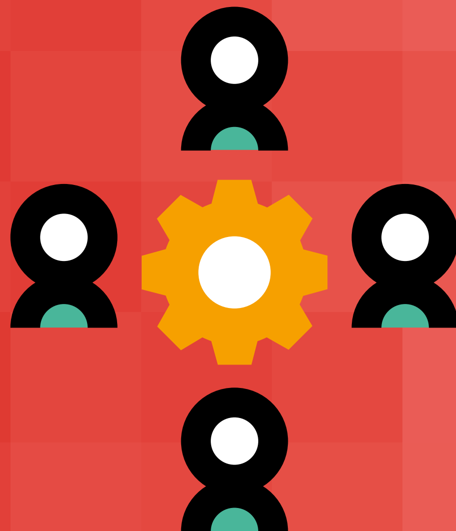


Ce diagramme indique l'importance respective de chaque pilier et l'ordre dans lequel les commencer. Pour autant, il ne s'agit pas de phases séquentielles mais bien de travaux en plusieurs parties et surtout plusieurs ébauches : une première passe de *collaborative modeling* débouche sur du *strategic design* venant challenger le modèle, pour l'affiner, explorer d'autres pistes, voire le refaire… Ces piliers forment un tout riche et cohérent.

Nous sommes dans un **processus** typiquement **agile** : on recherche les feedbacks rapides des utilisateurs et des experts du domaine pour éprouver l'avancement dans la conception et la réalisation de la solution.

Fil rouge

Le recrutement chez SOAT sera notre fil conducteur tout au long de ce livre blanc. Le besoin est d'aider les Ressources Humaines à trouver et recruter des profils de manière plus facile et efficace.



01 Collaborative modeling

Model-driven engineering

Le DDD fait partie des méthodologies d'ingénierie pilotées par une **modélisation** de problèmes métier. Le modèle est la clé pour réussir à construire une solution lorsque le domaine est complexe[6]. Le modèle vise donc à satisfaire les cas d'utilisation métier, à éliminer des contraintes ou à s'ouvrir à des opportunités. La particularité du DDD est d'être agnostique tant aux méthodes de modélisation qu'aux moyens techniques de réalisation.

Bird's eye view design

L'un des objectifs de la modélisation est d'aider à la **conception** d'un système. Nous reviendrons en détail sur la conception dans les deux chapitres suivants, consacrés aux deux autres piliers. Ce qu'il faut retenir ici est que la modélisation se fait de haut en bas, vers une granularité de plus en plus fine. On parle de *bird's eye view design*, expression qui se comprend en visualisant la scène suivante :

1. Un pigeon est en train de voler et voit deux points noirs autour d'un carré.
2. Le pigeon descend un peu et se pose sur un arbre. Il distingue désormais des carrés noirs et blancs entre les deux disques noirs.
3. Le pigeon descend encore et voit des figurines dans les cases noires et blanches.
4. Finalement il se pose sur le bord de la table, voit le plateau d'un jeu d'échecs et observe comment les deux personnes jouent. Il peut ainsi apprendre les règles du jeu, le métier.

La conception se passe de la même manière. On commence par modéliser grossièrement le système en entier puis on modélise les sous-systèmes, les applications, les librairies, les services, etc. Les modèles utiles sont ceux qui se construisent petit à petit, en passant d'une vision de haut niveau aux détails les plus fins. Ces différents niveaux en perspective permettent de produire une solution globale plutôt qu'une optimisation locale.

Modèle du domaine

Quand on parle de modèle, on ne désigne pas spécifiquement un diagramme UML ou tout autre artéfact. Un modèle est avant tout un ensemble d'abstractions. **L'abstraction** consiste à sélectionner les éléments en relation avec le problème métier et à les organiser. Tous les autres éléments sont délibérément écartés par souci de **simplification**. De même, on ne sélectionne que les détails pertinents dans les éléments et seules certaines de leurs interactions sont modélisées.

La réalité ne peut pas être modélisée en détail sans une explosion de la complexité, cheval de bataille du DDD. On reste donc focalisés sur les **cas d'utilisation du domaine métier**, lui-même suffisamment complexe pour amortir l'investissement dans le DDD par les bénéfices qu'il permet. C'est pourquoi l'on parle de **modèle du domaine** (*domain model*).

6. Complexe selon la terminologie du framework Cynefin https://en.wikipedia.org/wiki/Cynefin_framework

*«**All** models are **wrong**. **Some** are **useful**.»*

George Box[7]

Un bon modèle satisfait les cas d'utilisation métier et peut aussi enlever des contraintes ou amener de nouvelles opportunités. Trouver un tel modèle est toujours difficile et demande beaucoup d'expérience et d'expérimentations. Il ne faut donc pas hésiter à challenger en permanence son modèle, en particulier lors de la prise en compte de nouveaux besoins métier.

Exemple de modèle

L'exemple classique de modèle est une carte des transports en commun d'une ville :

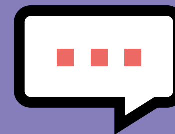
- **Problem** *(Use case)*
  Aider les clients à établir un itinéraire entre deux stations
- **Domain**
  Réseau des transports
- **Domain model**
  Carte des transports

Le modèle n'a plus qu'une légère ressemblance avec la réalité, le vrai tracé des voies. Les tracés sont rectilignes, aux couleurs des lignes de métro. Le placement des stations est ainsi pour faciliter la lecture plutôt que pour représenter la distance réelle, détail qui n'aide pas à solutionner le problème.

Sur d'autres cartes vont figurer certains monuments touristiques car elles intègrent un autre besoin métier, en relation avec le tourisme.

7. https://fr.wikipedia.org/wiki/George_Box

Mélanger plusieurs métiers est un exercice délicat. Il est plutôt recommandé de faire **un modèle par domaine métier**, même pour des cas d'utilisation similaires.



Artéfacts

Le modèle est abstrait. Il est mental, dans les têtes des personnes impliquées dans la réalisation du système. On peut le concrétiser en artefacts de différentes formes : diagrammes, schémas, documentation... Les supports sont eux-aussi variés : à main levée sur du papier, au tableau blanc, avec des post-its ou enfin au format informatique.

Cela facilite la compréhension du métier, complexe, en construction ou en évolution, mais aussi le partage des connaissances. Cela aide à faire en sorte que tout le monde soit **aligné**, en phase. Ces artéfacts peuvent également servir à l'élaboration du modèle, à sa compréhension et à sa diffusion.

Il faut néanmoins **éviter de s'attacher** à de tels artéfacts. Des artéfacts formels sont coûteux à produire, donc à maintenir. Par conséquent, personne ne le fera correctement. Ils vont donc rapidement finir par ne plus être fidèles au modèle qui ne cesse d'évoluer comme les besoins métier. Dès lors, l'existence d'artefacts faux fait prendre de sérieux risques. En s'appuyant sur ceux-ci, on peut prendre des décisions erronées et potentiellement lourdes de conséquences.

Le principe **YAGNI** *(You Ain't Gonna Need It)* dont on parle pour le code s'applique également ici. On a rarement un réel besoin de coucher un modèle au propre. Mieux vaut garder en tête qu'ils sont éphémères, **jetables**. C'est pourquoi il est tout à fait adapté de les réaliser sur une feuille volante ou sur un tableau blanc.

Parallèle avec les estimations

On connaît la difficulté, et même l'impossibilité dans l'ingénierie logicielle, de réaliser des estimations, des chiffrages en jours/homme ou en points de complexité, fiables. Pour autant, cela peut avoir des bénéfices que l'on manquerait en refusant tout travail d'estimation. L'important n'est pas de produire les artefacts de modèle (chiffrages). L'important se situe dans les sessions de modélisation (d'estimation) faites de manière **collaborative.** Tout le monde participe, ce qui permet de bénéficier d'un effet de groupe *(d'une intelligence collective ?)* et du partage de connaissances du problème métier, de l'ébauche de solution envisagée et des possibles difficultés techniques.

Analysis model, code model

Une fois seulement trouvé un modèle d'analyse *(analysis model)* qui est utile, on peut commencer le modèle implémenté dans le code *(code model)* mais ce n'est pas séquentiel. Le processus est itératif, le travail s'effectue sur les deux modèles, en recherchant du feedback de l'un vers l'autre. Cela permet aussi de varier les perspectives pour aider l'exploration du domaine.

Les deux types de modèles sont fortement liés par le langage commun *(ubiquitous language)* que nous allons expliquer par la suite. Les découvertes peuvent se faire de part et d'autre, puis la connaissance est partagée, les modèles synchronisés et le modèle général enrichi. En liant étroitement le code à un modèle sous-jacent, on a non seulement un code qui fait sens mais aussi un modèle rendu pertinent. Ce lien doit aussi être évident. Pour cela, on privilégie un modèle qui s'implémente le plus naturellement possible et un design qui reflète le modèle de manière très littérale.

Code et modélisation

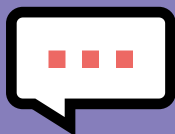
Le seul artéfact qui va concrétiser le modèle aussi fidèlement que possible et surtout dans la durée est le **code source.** En bout de chaîne, le code qui a servi à produire le système est la source de **vérité.** Le code n'est pas le produit final mais une spécification confiée au compilateur qui va construire le produit.

Le code peut également servir en amont de la réalisation. L'ingénierie logicielle a la particularité d'avoir un délai de réalisation (ici de compilation) extrêmement rapide comparativement aux autres ingénieries. Autant s'en servir afin d'obtenir des feedbacks plus tôt.

On peut ainsi réaliser des **prototypes** qui permettent d'évaluer la faisabilité de notre modèle et les interactions dans notre système. Tobias Goeschel anime un atelier *Exploring your Design with Domain Prototyping*[8] qui permet entre autres d'expérimenter le prototypage.

Ces prototypes, comme les autres artéfacts concrétisant le modèle, sont jetables.

> « ***Experiment in code*** *to prove the usefulness of the model and to give feedback on the compromises that a model needs to make for technical reasons.* »
>
> Scott Millet[9]



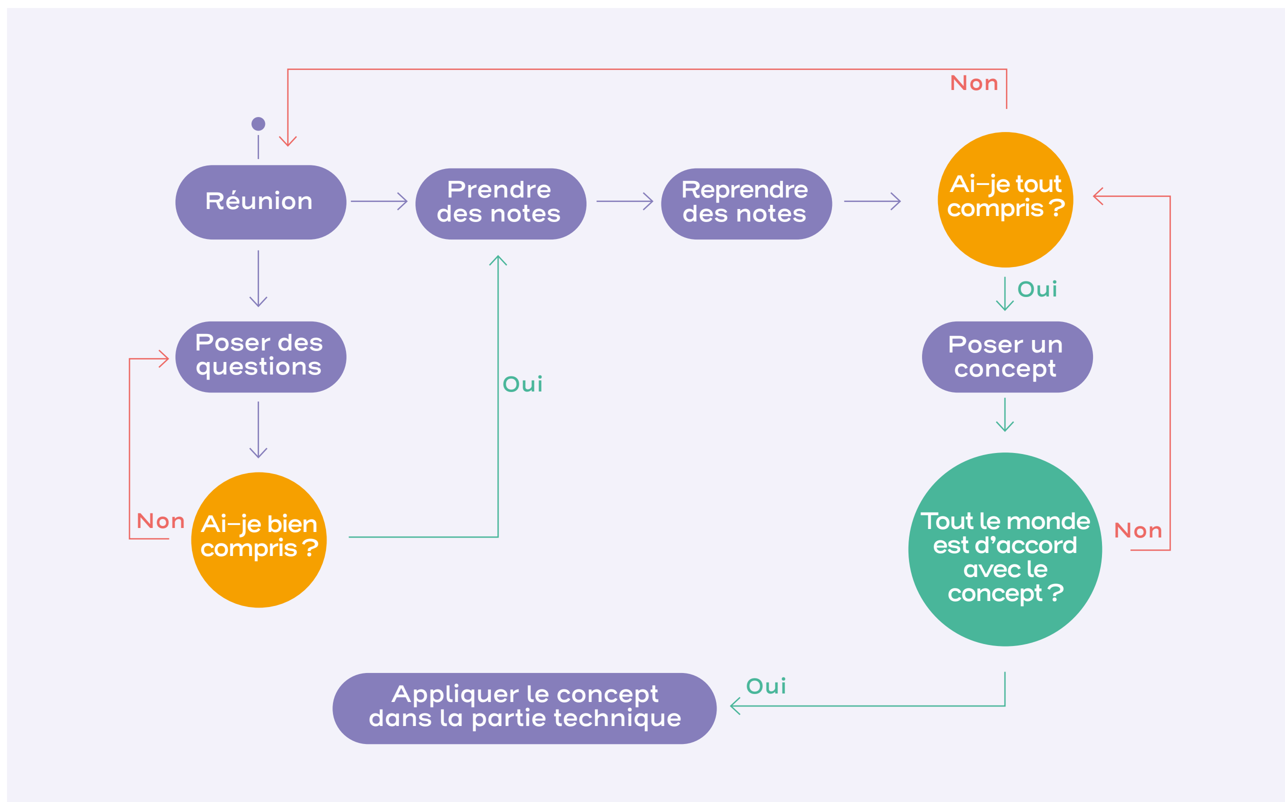
Savoir supprimer du code est une étape **majeure** dans la carrière d'un développeur.

Il est également possible d'écrire le modèle directement en code, dans les types, dans un langage de programmation suffisamment concis pour que cela soit aisé. Le **F#** en est un bon exemple, dont Scott Wlaschin est un fervent promoteur[10].

8. https://dddeurope.com/2019/speakers/tobias-goeschel/
9. Auteur du livre *Patterns, Principles and Practices of Domain-Driven Design* (2015)
10. Cf. livre *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*, 2018

Importance de la communication

Pour renforcer la collaboration lors de la modélisation, le DDD met fortement l'accent sur la communication. Cette communication accentue les chances d'une compréhension partagée et ainsi d'une solution adaptée aux problèmes. Le schéma suivant propose un processus communicationnel qui aide à mieux comprendre les besoins du métier :



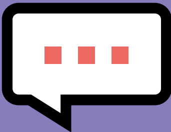
De prime abord, cela paraît lourd et long. Or, une erreur de conception de logiciel est d'autant plus coûteuse à corriger qu'elle est constatée tardivement (cf. diagramme suivant), et bien plus lourde et difficile que ce processus. C'est donc un mal pour un bien. Une fois la communication rodée, on peut passer à des échanges plus naturels, tant qu'ils restent riches.

Coûts

Développement
Tests manuels
Recette
Production
Temps

Langage commun

Le langage est la clé d'une bonne communication. Si des personnes ne parlent pas la même langue, elles auront du mal à communiquer. Même en parlant la même langue, les mots peuvent avoir plusieurs significations. Seul le contexte permet de comprendre pleinement et sans ambiguïté conversations et écrits.



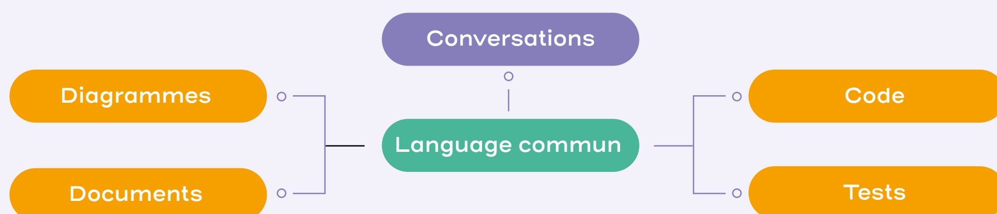
L'un des objectifs du DDD est la mise en place de l'*ubiquitous language*. Nous utiliserons l'expression **langage commun**, traduction non fidèle, *ubiquitous* voulant dire omniprésent, mais plus facile à appréhender, moins source de confusion à notre avis.

Les expressions «langage commun» et «langage omniprésent» sont toutes les deux réductrices, ce langage revêtant de multiples facettes :

- **Métier** et **unique** : il s'agit de termes et d'expressions en relation directe avec le problème à solutionner, en bonne partie déjà employés par le métier mais aussi sélectionnés ou créés pour désigner de nouveaux éléments issus de la modélisation de solutions.
  Les termes techniques n'en font pas partie, de manière à se concentrer sur le métier et ne pas mélanger les préoccupations, ce qui serait source de complexité. Par exemple le métier ne parle pas SQL, HTTP ou Kafka.
- **Spécifique** et **non ambigu** : chaque mot est précis et univoque. Si un mot a plusieurs sens, un seul sens sera employé. S'il existe plusieurs termes pour désigner la même chose, un seul sera utilisé de manière exclusive.
- **Contextuel** : en corollaire, le langage ne peut être compris pleinement et sans ambiguïté que dans un contexte donné, celui du domaine en question.
- **Commun** et **partagé** : le langage est parlé par tous les participants du projet, des sachants métier à l'équipe de développement.
- **Omniprésent** et **fédérateur** en étant utilisé partout et par tous : dans les conversations, les cas d'utilisation, le modèle, les *user stories*, le code, les tests, la documentation...

Le langage commun agit comme un liant, fédérant toutes les expressions du modèle :

Modèle de domaine



Les acronymes

Ils permettent certes une vélocité dans la communication, mais ils peuvent aussi représenter une surcharge mentale pour revenir à l'expression complète derrière l'acronyme. Notre recommandation est d'essayer de s'en passer.

Il ne s'agit pas d'être rigoriste. On cherche à faciliter l'usage du langage commun. On accepte volontiers les termes familiers, par exemple *Propale* pour signifier proposition d'emploi.

Ce travail de sémantique est délicat. On plonge au cœur de la représentation que chacun se fait du métier et de la solution. Une formule d'Alfred Korzybski[11] nous le rappelle : « La carte n'est pas le territoire qu'elle représente ». Les différences sont parfois significatives entre plusieurs personnes, et cela parasite la communication. Ce travail sur le langage est donc nécessaire pour viser une communication optimale.

Prendre soin du langage commun

Un tel langage permet une communication efficace. Mais il représente une contrainte importante : il faut beaucoup de discipline et de vigilance pour utiliser toujours le même terme pour exprimer le même concept. C'est aussi l'un des facteurs déterminants pour réduire la complexité due à l'ambiguïté du langage. Il faut donc une bonne communication également en amont pour expliquer le langage commun et obtenir l'adhésion de tous.

La mise en place du langage commun passe par la construction d'un **glossaire** qui explicite chaque terme et indique les éventuels synonymes proscrits. Si le métier est vaste, composé de plusieurs sous-domaines, il faut faire ce travail pour chaque sous-domaine[12]. À mesure que l'on accumule des connaissances métier pertinentes pour le projet, que l'on découvre des concepts, on enrichit glossaire et langage commun.

Cela nécessite une certaine **discipline**, car ce travail n'est possible qu'avec la participation active de tous les membres du projet : lorsque quelqu'un n'emploie pas un terme du langage commun, il convient de le signaler et d'agir en conséquence : soit il s'agit d'un nouveau concept du domaine à faire entrer dans le langage commun, soit cela correspond à un concept existant et il faut alors utiliser le terme défini.

Fil rouge
Exemple de glossaire dans le domaine du recrutement chez SOAT

TERME	DÉFINITION
Candidat	Personne postulant chez SOAT. Il passera plusieurs entretiens pour éventuellement devenir un consultant SOAT
Consultant Recruteur	Consultant SOAT volontaire et habilité à faire passer des entretiens techniques à des candidats
Point HowSo	Point qui correspond à une action de capitalisation effectuée par un consultant SOAT

11. https://fr.wikipedia.org/wiki/Alfred_Korzybski
12. L'alignement "un sous-domaine == un contexte" est un cas idéal. Dans la pratique, les sous-domaines ne sont pas tous bien délimités et peuvent correspondre chacun à plusieurs contextes dont les limites sont définies (cf. *Bounded Context*). On a un langage commun par contexte. Un tel sous-domaine aura donc plusieurs *langages communs*.

Knowledge crunching

Le *knowledge crunching* est la collecte sélective des connaissances métier. Ce travail est essentiel pour mettre en place le langage commun et produire un modèle de domaine pertinent et efficace.

Où se trouvent ces connaissances ? Éventuellement dans de la documentation, mais encore faut-il qu'elle soit suffisamment claire, complète et à jour. En fait, ces connaissances sont souvent non formalisées et se trouvent essentiellement dans la tête des **experts du domaine** (domain experts), c'est-à-dire toute personne qui peut offrir une meilleure compréhension du problème : utilisateurs, *product owners (PO), business analysts (BA)*, autres équipes techniques.
Les parties prenantes du projet qui fournissent l'expression du besoin ne sont pas forcément les plus à même de répondre à des questions précises sur le domaine.

Il faut mettre à profit les experts métier dès que l'on s'attaque à la modélisation des parties **les plus critiques ou les plus complexes.** Il ne faut pas se contenter de leur regard extérieur ou de leur avis et validation a posteriori, mais les impliquer dans la modélisation même.

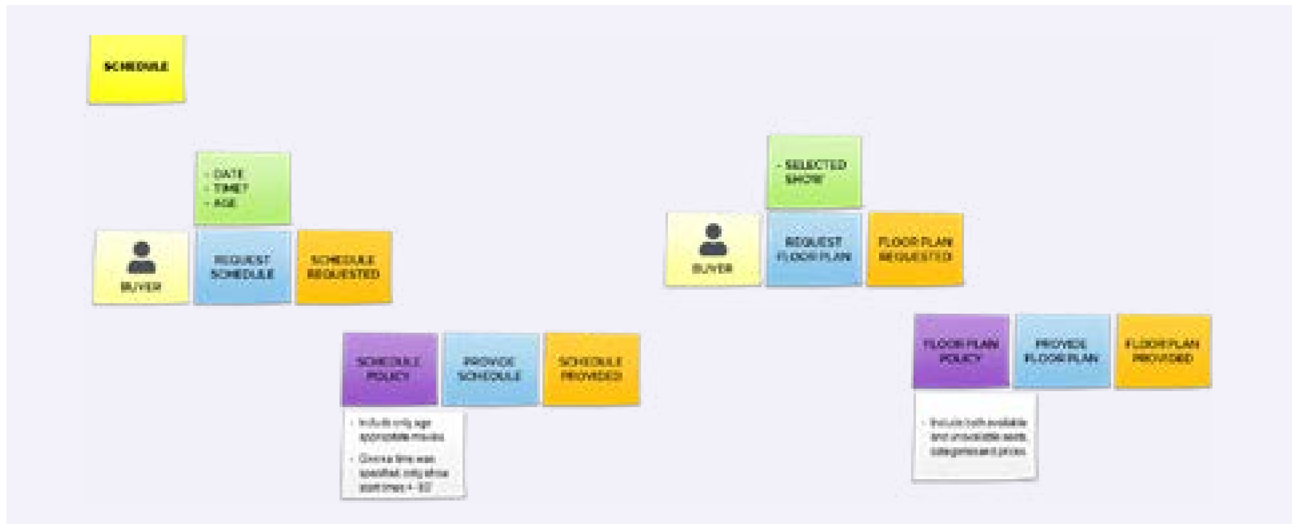
Le *knowledge crunching* requiert du temps. Or, les experts métier sont généralement des gens déjà bien occupés. Des sessions de travail formelles seraient trop rébarbatives et conduiraient à des défections. La clé est de rendre les ateliers **attractifs** et un soupçon **fun.**

Il existe différents ateliers types pour le *knowledge crunching*. L'*Event Storming* et l'*Example Mapping* se démarquent actuellement.

Event Storming

L'*Event Storming* est un atelier de modélisation et de *knowledge crunching.* Il permet dans un temps court d'extraire le maximum de connaissances sur un sujet. Il a été inventé par Alberto Brandolini[13]. Son format le rend facile à mettre en œuvre car il ne nécessite pas de lourds moyens logistiques : ni ordinateur, ni rétroprojecteur, ni téléphone. Il suffit simplement d'un **vaste mur** couvert d'une frise de papier blanc (par exemple des A2) et de post-it.

13. https://www.eventstorming.com/



En revanche, malgré la difficulté majeure que cela représente, **le présentiel (physiquement sur place, ou bien à distance via des outils adaptés qui nous fournissent un espace infini et des post-it, comme Miro) est obligatoire** : il faut rassembler le maximum d'acteurs du projet, venant de tous horizons pour aller au-delà des éventuels silos organisationnels.

Cet atelier se fait de manière collaborative en **plusieurs étapes.** Le processus métier est d'abord «éclaté» en événements de domaine, que chacun écrit sur des post-it orange et les pose sur le mur de manière **chronologique** : à gauche le début du process, à droite sa fin. Le mur représente donc la ligne du temps. Les participants discutent ensuite pour faire le tri entre des post-it redondants, les déplacer, en trouver des manquants...

On peut ensuite enrichir la vision du process avec les commandes (l'action qui déclenche un événement) et les acteurs (humain ou machine) à l'origine des événements, les données clés, les questions/incertitudes restantes, etc.

L'avantage des **post-it** est de pouvoir être repositionnés, enlevés, corrigés et réécrits très facilement. On garde un petit côté feuille volante mais qui est délibéré : l'objectif se situe dans les conversations, les découvertes, le partage de connaissances... Au besoin, on peut prendre en photo le résultat ou conserver les feuilles pour une autre session d'approfondissement.

Au fur et à mesure de l'avancement de l'atelier, on peut voir que certaines parties du système ressortent et forment naturellement un ensemble cohérent. Cela nous aide à **découper notre système** afin de le rendre moins complexe et à se focaliser sur des parties plus petites. L'*Event Storming* peut d'ailleurs être fait sur l'ensemble d'un système ou sur une plus petite partie.

Behaviour Driven Development

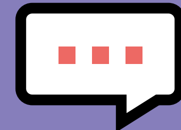
Le BDD est issu d'une réaction à certaines dérives du TDD lorsque les gens sont trop focalisés sur des détails techniques. Cela pose déjà des problèmes d'ordre technique comme la fragilité des tests qui cassent à la moindre modification du code de production. Le BDD adresse **un problème plus ennuyeux : l'oubli du métier !** Il permet de se recentrer sur le métier et ses cas d'utilisation. On conserve un objectif de test, de validation de la réalisation d'une fonctionnalité qui se trouve dans le TDD historique (celui de Kent Beck[14] entre autres).

Le BDD comprend deux facettes : la **découverte délibérée** (*deliberate discovery*) et bien sûr le TDD.

C'est à cette première que nous allons porter notre attention ici. L'idée de base repose sur le fait que l'on découvre souvent trop tard des éléments clés. Cette découverte tardive nécessite alors de réviser sa copie et de refaire une partie du travail. Il s'agit donc d'aller délibérément à la découverte des inconnues.

Les développements en mode exploratoire sont une manière de cibler les inconnues techniques. Mais ce sont les inconnues fonctionnelles dont il est question ici. Une des manières les plus efficaces de partir à cette pêche aux informations repose sur deux facteurs : **conversation** et **collaboration**.

Les équipes les plus productives sont celles qui cherchent délibérément à découvrir ce qu'elles ignorent, ceci avant même le début des développements.



Des sessions collaboratives

L'essence même du BDD prend forme dans des sessions "en **3-amigos**". On y réunit représentants du métier (product owners, experts métier, utilisateurs), équipe de développement et testeurs. À ces 3 amigos principaux, on peut convier également des Ops, des designers UX, etc. Cette variété de profils signifie différentes perspectives, ce qui permet des échanges riches, ponctués de découvertes sur le domaine du problème. Cela permet aussi de mettre tout le monde au même niveau de compréhension et de s'accorder sur la manière de qualifier et de valider les développements attendus.

L'objectif de ces sessions est de définir les **critères d'acceptation** des user stories. Ces critères peuvent être écrits en utilisant le formalisme du langage Gherkin *(Given/When/Then)*[15] et doivent être sous forme d'exemples concrets décrivant comment le logiciel devrait se comporter pour répondre à un besoin métier et satisfaire des règles métier. C'est pourquoi l'on parle aussi de **spécifications par les exemples**[16].

Ces spécifications servent de socle pour l'accord commun. L'équipe de développement peut les intégrer dans ses "définitions" :

14. https://fr.wikipedia.org/wiki/Kent_Beck
15. https://docs.cucumber.io/gherkin/
16. Cf. livre *Specification by Example*, de Gojko Adzic, 2011

Definition of Ready	Qualifie qu'une user story est « prête » à être développée[17].
Definition of Done	Indique qu'une user story est « finie » pour l'équipe, par exemple : « prêt pour les testeurs », « prêt pour le déploiement », « livré en production »[18].

Ces spécifications par les exemples peuvent également servir de matière pour les réalisations des participants du 3-amigos : spécifications détaillées, code, cahier de tests, tests automatisés…

Test Driven Development (TDD)

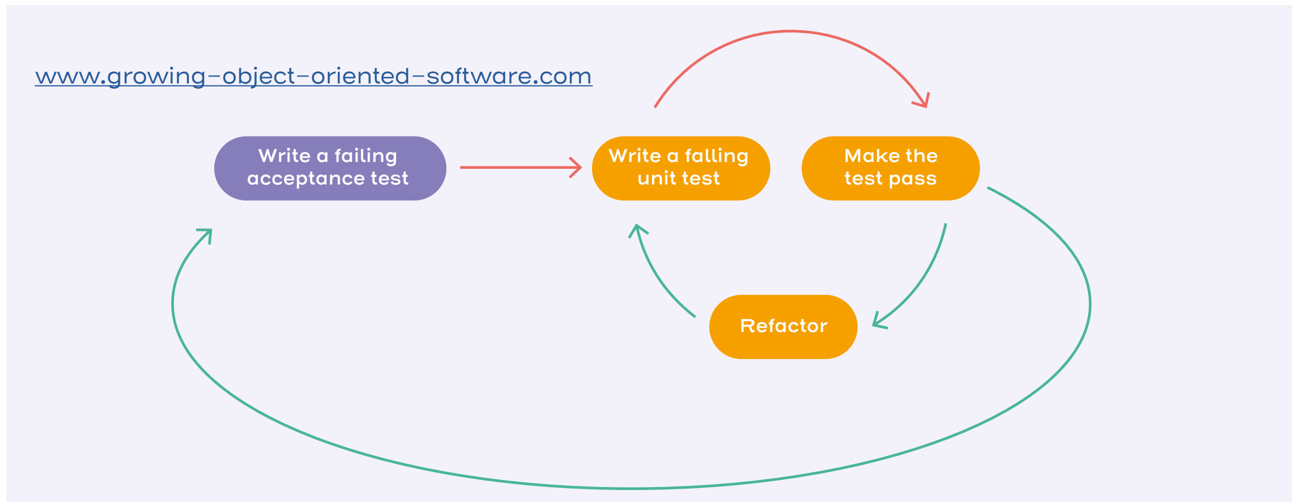
Le code sera écrit en **TDD** si l'on veut être totalement en accord avec le BDD.

Dans l'esprit original, l'objectif n'est pas d'écrire des tests mais de piloter le développement guidé par les tests. Le TDD à la sauce BDD fait usage d'un langage naturel pour décrire les tests[19].

Le TDD peut se pratiquer à différents niveaux de granularité :

- Test d'acceptation (TA) global, sous la forme d'un test d'intégration (TI), d'un test sous-cutané ou d'un test de bout en bout[20].
- Tests unitaires (TU).

On parle alors de double boucle TA/TU qui va généralement de pair avec l'approche *outside-in* de l'école londonienne du TDD.



17. https://www.scrum.org/resources/blog/walking-through-definition-ready
18. https://www.scrum.org/resources/blog/done-understanding-definition-done
19. Exemple de tournure de phrases en anglais : *[The system under test] should (do something) when (this thing happens) given (it is in that state).*
20. On parle aussi d'ATDD, A pour *Acceptance*

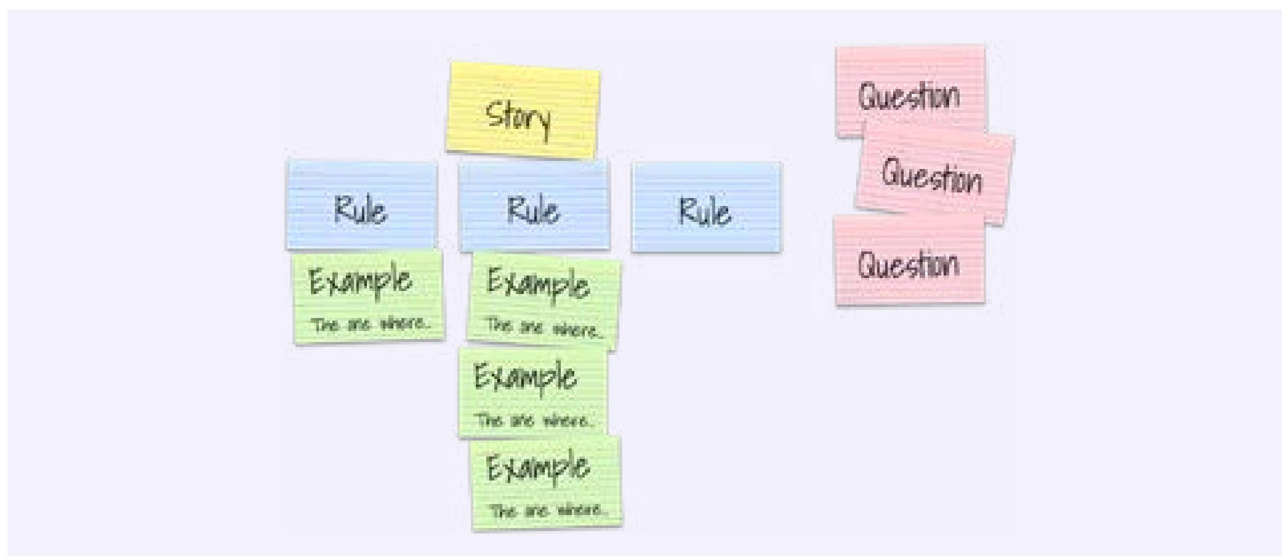
Automatiser les tests

Il faut bien faire **attention** à ne pas penser que le BDD signifie automatiser les tests. Les développeurs peuvent venir au BDD par le biais de frameworks d'automatisation de tests comme Cucumber, FitNess[21] ou encore en pratiquant le TDD *outside-in* avec double boucle TA/TU. Or, l'essence du BDD est bien le focus sur le métier. En cela, il s'accorde parfaitement avec le DDD.

Example Mapping

Il s'agit d'un atelier permettant de faciliter le BDD, qui se pratique sur des **fiches bristol** ou à l'aide de post-it comme l'*Event Storming*. On peut d'ailleurs les combiner dans une même session pour attaquer le problème sous différents angles.

L'idée est de structurer la recherche d'exemples à l'aide des *Stories* chapeau et des *Règles*, support des critères d'acceptation. Les exemples viennent **illustrer les règles.** Les éventuelles incertitudes restantes sont consignées sous forme de *Questions*.



Beaucoup de choses peuvent émerger d'un tel atelier : nouvelles règles, nouvelles stories... Le principal résultat ne sera pas forcément écrit : c'est la compréhension partagée entre les différents participants qui compte.

Plus d'informations sur le blog de Cucumber[22] duquel est extraite l'image ci-dessus.

21. http://docs.fitnesse.org/FrontPage
22. https://cucumber.io/blog/example-mapping-introduction/

Autres outils

Il existe beaucoup d'autres outils et techniques qui dépassent le cadre de ce livre blanc :

- *Business Model Canvas*[23]
- Diagrammes de cas d'utilisation[24]
- *Impact Mapping*[25]
- *User Story Mapping*[26] de Jeff Patton
- *DDD Whirpool* d'Eric Evans[27 28]
- *Wardley Mapping*[29]
- *Domain Storytelling*[30]

Nous laissons le soin aux lecteurs de creuser ces sujets afin de diversifier leurs techniques de *knowledge crunching*.

Trucs et astuces pour l'animation

Le *knowledge crunching* nécessite des talents d'organisation, d'animation, de communication. Voici quelques trucs et astuces qui peuvent vous aider ou vous inspirer en tant qu'animateur/ facilitateur.

- Faire en sorte que tout le monde participe.
  - Faire attention aux personnes qui se détachent et s'isolent, tenter de les réimpliquer.
  - Vérifier que tout le monde est à l'aise et bénéficie d'une place dans le groupe. Ne pas hésiter à restructurer certains groupes déjà formés.
- Veiller à ne pas trop s'impliquer.
  - Faire attention à son propre placement par rapport aux participants : ne pas rester trop longtemps tourné vers un tableau.
  - Rester légèrement en retrait et prendre de la hauteur. Cela permet d'assurer l'animation de manière plus continue et de ne pas perdre de l'énergie à essayer de tout contrôler.
- Prêter l'oreille aux échanges.
  - Deux travers vont limiter les résultats d'un atelier, chacun situé à une extrémité du spectre : les détails insignifiants (cf. *bike-shedding*[31]) et les concepts trop abstraits ou flous.
  - Savoir orienter une conversation pour rétablir un juste équilibre abstraction/détails.
  - L'introduction d'exemples concrets peut parfois aider.
  - En milieu de session, repérer les parties lacunaires et chercher à combler ces trous : faire dérouler un scénario concret de bout en bout, poser des questions puissantes[32].
- Favoriser le questionnement.
- Inviter à faire des pauses régulièrement.
- Prévoir à boire et à manger.

---

23. https://en.wikipedia.org/wiki/Business_Model_Canvas
24. https://fr.wikipedia.org/wiki/Diagramme_de_cas_d%27utilisation
25. https://www.impactmapping.org/
26. https://www.jpattonassociates.com/user-story-mapping/
27. http://domainlanguage.com/ddd/whirlpool/
28. https://baasie.com/2019/02/04/model-exploration-whirlpool-domain-driven-design-the-first-15-years/
29. https://learnwardleymapping.com/
30. https://domainstorytelling.org/
31. Loi de futilité de Parkinson : https://en.wikipedia.org/wiki/Law_of_triviality
32. «Que veux-tu dire par ... ?», «Si l'on supprime cette partie que va-t-il arriver à ... ?» sont des exemples de questions puissantes

- Prévoir de l'espace suffisant pour tous les participants.
  - En corollaire : ne pas inviter trop de monde pour la salle en question
- Inviter les « bonnes personnes ».
- Ne pas donner de réponse mais guider les participants pour arriver à une solution.
- Planifier peut-être plusieurs séances si le sujet à traiter est trop gros.
- Préférer écourter un atelier pour ne pas épuiser les participants.
- Prévoir un *ice breaker* au début pour mettre tout le monde à l'aise.

Modélisation itérative et incrémentale

« Sophisticated domain models seldom turn out useful except when developed through an iterative process of refactoring, including close involvement of the domain experts with developers interested in learning about the domain. »

Eric Evans

La modélisation se déroule de manière collaborative, en élargissant le spectre à tout type de profil allant d'experts métier aux développeurs. Cela nécessite généralement beaucoup de temps dont ne disposent pas toujours certains profils. La modélisation s'effectue alors en plusieurs fois, de manière itérative et incrémentale :

Incréments

Le modèle peut être construit par parties :

1. Au tout départ, on peut chercher à avoir une **vision d'ensemble** du modèle, en prenant en compte l'existant : les relations avec les autres modèles, les autres équipes, les autres systèmes.
2. Ensuite, les détails se matérialisent en modélisant les **cas d'utilisation** un par un, en commençant par les plus importants et en s'appuyant au maximum sur des scénarios concrets.
3. Il y a aussi les incréments imprévus, issus de la découverte de concepts ou d'approfondissements de portions déjà modélisées.

Les sessions peuvent avoir un côté **exploratoire**, en particulier lorsque le modèle est incomplet ou incohérent. Ces sessions ou parties de sessions ne seront pas forcément productives (en quantité) mais permettront l'émergence de concepts à la fois simples et très puissants, supports d'un modèle lisible, expressif, qualités également attendues dans le code.

Itérations

Il est préférable de prévoir plusieurs sessions collaboratives espacées dans le temps pour qu'elles ne soient **pas trop longues** et donc pas trop fatigantes. Il faut aussi pouvoir disposer des experts métier dont les calendriers ne permettent pas de créneaux trop longs. Les différents incréments prévus peuvent alors être passés en revue chacun dans une session différente.

Entre plusieurs itérations, le modèle peut mûrir dans la tête de chacun, le temps faisant son effet, ou à la suite d'expérimentations de l'équipe de développement. Plus on joue avec le modèle, plus on comprend le domaine et plus on peut affiner le modèle. Il ne faut jamais arrêter la modélisation au premier jet. Même si l'on en est déjà satisfait, le modèle n'en est qu'à l'état de brouillon. On aura donc également des sessions permettant d'améliorer des portions existantes du modèle.

Il est possible et même conseillé d'évaluer plusieurs modèles en même temps. Pour ce faire, au fur et à mesure de l'exploration du problème, on constitue un ensemble de scénarios de référence. Chaque scénario est décrit en utilisant successivement le langage et la logique de chaque modèle. Plus le résultat est simple et meilleur sera le modèle. Mais la difficulté est de ne pas écarter trop tôt un modèle pour l'heure moins élégant mais qui s'avèrera un meilleur compromis pour traiter de nouveaux scénarios, ceux-ci rendant alors bancal ou inadapté le modèle actuellement favori.

Décomposition en sous-domaines

L'un des objectifs les plus importants de DDD est de diminuer la complexité afin de la garder sous contrôle, à un niveau abordable par les acteurs d'un projet.

Une stratégie consiste à découper en plus petites parties le domaine métier et à s'attaquer à des sous-problèmes indépendamment. On diminue alors la combinatoire. Ajouter de nouveaux comportements au système existant devient plus facile et l'on risque de générer moins de régressions sur les autres parties du système rendues indépendantes, découplées.

Les diagrammes de cas d'utilisation et l'*Event Storming* peuvent nous aider à trouver les différents sous-domaines d'un système.

Toutes les parties d'un système n'ont pas la même importance. Le DDD permet d'identifier trois types de sous-domaines : ***Core***, ***Generic*** et ***Supportive***.

Core Sub-Domain

Le sous-domaine principal est la partie du système qui fait la différence par rapport aux concurrents. C'est la "sauce maison" de l'entreprise, la raison d'être du produit.

Sauf exceptions, le Core Domain sera au cœur des préoccupations pendant plusieurs années : 2, 5, 10 ans, qui sait ? Tant que cela reste une stratégie payante. C'est là qu'il faut concentrer ses efforts, en particulier tout ce qui va faciliter la maintenance du produit. Plus facilement on sera en mesure de faire évoluer le produit, plus rapidement on livrera de nouvelles fonctionnalités, ce qui est un facteur déterminant pour garder une longueur d'avance sur la concurrence, ou avoir un retour rapide sur certains risques.

C'est pourquoi il est stratégique d'avoir sur cette partie du système des développeurs avec une qualité de développement irréprochable.

Generic Sub-Domain

Toute solution correspondant à un sous-domaine déjà existant et qui n'est pas propre à l'entreprise est de type *Generic*. La comptabilité est généralement dans ce cas-là. Plutôt que de spécifier en détail un logiciel pour ce sous-domaine, il est en généralement plus pertinent d'investir dans un produit du marché : progiciel, logiciel, librairie...

Supportive Sub-Domain

Tout sous-domaine qui n'est ni *Generic*, ni *Core* est de type *Supportive*. On peut s'aider de ces questions pour identifier un tel sous-domaine :

- Si l'on supprime ce sous-domaine, le système continuera-t-il de répondre aux problèmes métier ?
  Si ce n'est pas le cas, il s'agit d'un sous-domaine *Core*.
- Existe-t-il une solution qui répond aux besoins de ce sous-domaine, qu'elle soit gratuite ou payante ?
  Si oui, on s'oriente vers un sous-domaine *Generic*.
  - Sinon, il s'agit d'un sous-domaine *Supportive*.

Étudions quelques exemples pour mettre cela en pratique :

Fil rouge

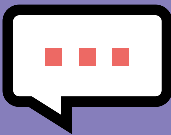
Dans le contexte de notre fil rouge, on peut identifier ces différents sous-domaines :

1. *Recrutement* : sans trop de surprise, c'est le sous-domaine *Core*.
2. *Communication* : cela peut être un sous-domaine *Generic* si SOAT décide d'utiliser un logiciel externe d'envoi de mails et d'invitations.
3. *Gestion des CV* : ce sous-domaine est *Supportive*. Son absence n'empêche pas le processus de recrutement. Malgré tout, c'est une solution propre au SI de recrutement chez SOAT.

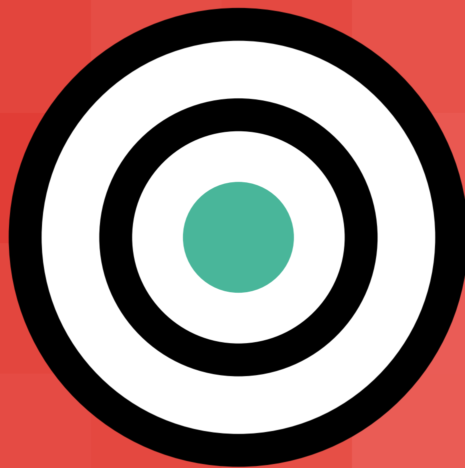
Constructeur automobile

On peut subdiviser ce domaine en trois parties : *Moteur, Amortissement et Habitacle*. En revanche, dire de quel type de sous-domaine il s'agit **dépend totalement du contexte.** Selon la marque et la stratégie du constructeur, le sous-domaine *Core* ne sera pas le même :

CONSTRUCTEUR	*MOTEUR*	*AMORTISSEMENT*	*HABITACLE*
Ferrari	✓		
Peugeot		✓	
Mercedes			✓
Rolls-Royce	✓	✓	✓



Les sous-domaines se trouvent dans l'espace du problème. Une fois qu'ils ont été découpés et que leur niveau d'importance pour le métier a été défini (*Core, Generic, Supportive*), notre objectif est de les modéliser pour passer dans l'espace de la solution. C'est là qu'entrent en jeu les parties stratégiques et tactiques.



02 Strategic design

La conception du système s'effectue à différentes échelles, d'où l'emploi du vocabulaire militaire avec les termes stratégie et tactique. La stratégie est de plus haut niveau : elle s'applique au système entier afin d'en définir les sous-éléments et leurs relations. La tactique porte sur les composantes de ces sous-éléments comme nous le verrons dans le chapitre suivant.

Dans le DDD, les sous-éléments du système à concevoir se nomment les *Bounded Contexts* et leurs relations sont définies grâce au *Context Mapping*.

Bounded Context

Définition

Le *Bounded Context* est un pattern central et original en DDD. C'est une notion qui peut paraître assez vague de prime abord. Les définitions que nous allons voir juste après sont en effet succinctes mais suffisantes : être plus précis serait être moins ouvert. C'est avec la pratique que la notion de *Bounded Context* se clarifie. Pour cela, nous poursuivrons l'étude de notre cas métier fil rouge.

Le **contexte** est une notion sémantique qu'Eric Evans définit ainsi :

« The setting in which a word or a statement appears that determines its meaning. ***Statements about a model can only be understood in a context.*** *»*

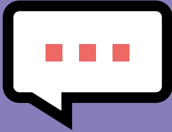
Eric Evans

Bounded Context et langage commun

On voit la relation étroite avec le langage commun. Le contexte donne un cadre d'application d'un langage commun.

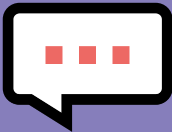
On parle de *contexte borné (Bounded Context)* de manière à rendre explicite et importante la notion sous-jacente de limites, de frontières. Ce sont typiquement les **frontières** d'un sous-système (frontières techniques) ou le travail d'une certaine équipe (frontières métier et organisationnelle).

Il s'agit également des frontières linguistiques d'application du langage commun. Nous avons vu en introduction de ce livre les effets de l'absence de langage commun, en particulier le fait qu'un terme puisse avoir plusieurs significations. À l'intérieur d'un *contexte borné*, nous n'avons qu'un seul langage commun.



Chaque terme n’a qu’une seule signification dans un contexte donné.

À l’échelle du domaine, un terme peut avoir plusieurs significations. Pour diminuer cette ambiguïté source de confusion et de complexité, on cherche à séparer les contextes d’utilisation. On aura plusieurs contextes et un langage commun par contexte.



Chaque *contexte borné* possède un glossaire, un domaine métier en comporte donc plusieurs.

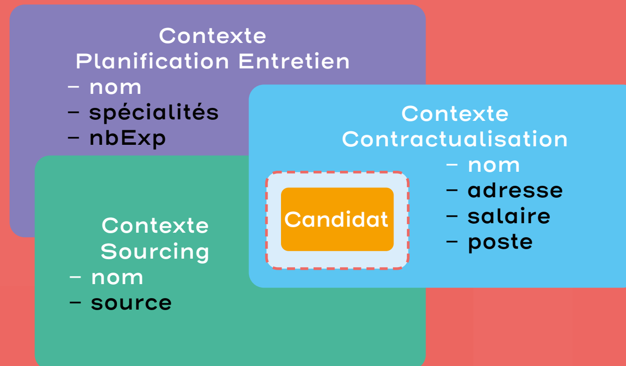
Étendue et intégrité du modèle

Le niveau stratégique du DDD consiste à découper un modèle trop vaste, comme sont délimités des pays dans un continent. Le niveau tactique se passe à des échelles plus fines, l’équivalent des échelles administratives d’un pays (région, département, ville, quartier).

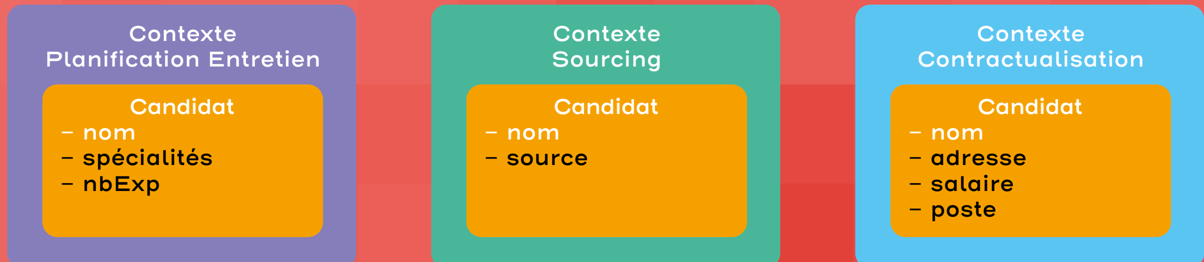
On cherche donc à définir un contexte borné le plus indépendant de l’extérieur possible. On cherche à éviter le *Big Ball of Mud* qui est le résultat de l’agrégation d’éléments disparates et entremêlés. Les concepts et les données extérieurs sont donc exclus du contexte, d’où le besoin de traduction entre contextes que l’on verra un peu plus loin.

Fil rouge

Revenons à notre fil rouge, le processus de recrutement chez SOAT. Considérons le terme *Candidat*. Au début, les seules informations nécessaires pour identifier un candidat sont son nom et la source qui a permis son entrée dans le processus. La prise en compte d'autres règles métier va enrichir le concept de *Candidat*. On va avoir besoin de certaines informations pour son entretien technique, et d'autres informations pour la phase de contractualisation. Le modèle devient alors plus complexe, hétérogène :



On sent le besoin de faire le tri, de ranger ces informations dans des tiroirs. La solution consiste à introduire des contextes bornés dans lesquels le *Candidat* regroupe des informations spécifiques à chaque contexte.



En s'intéressant au domaine métier et en sollicitant des experts métiers, nous réalisons désormais que le terme *Candidat* n'est pas approprié partout. Un nom plus précis peut être trouvé pour chaque contexte.

Bounded Context et Event Storming

Un outil performant pour trouver les contextes bornés est l'*Event Storming* dont nous avons déjà parlé. Après une ou plusieurs sessions, on peut constater différents groupes de post-it qui forment entre eux **une unité logique**. Ces groupes sont susceptibles de former des contextes bornés.

Contexte et organisation

En matière d'organisation, on préférera qu'une seule équipe de développement travaille sur un contexte borné. Elle pourra ainsi le faire de manière autonome. Si plusieurs équipes travaillent sur le même contexte, garantir la synergie entre les équipes est vain.

Les limites doivent être également définies clairement au niveau technique. Qu'il s'agisse d'une application, d'une librairie ou de services, il est préférable d'isoler le code sur un dépôt à part, *repository git* ou autre, de même que tout le pipeline d'intégration et de déploiement continus *(CI/CD)*. Autrement, le risque de «contamination» avec d'autres projets est grand.

Idéalement, chaque sous-domaine est associé à un seul contexte borné, mais cette règle est difficile à respecter dans la pratique et dans le temps. Les réorganisations d'équipes, métier comme techniques, sont monnaie courante. Cela provoque généralement une rupture de la relation 1-1.

Context Mapping

Le contexte borné définit l'intérieur et les limites d'un modèle. Or, il est rare d'avoir un contexte en isolation totale. Un contexte borné se situe généralement au sein d'un écosystème plus vaste avec lequel il interagit : les autres contextes du modèle, les systèmes existants souvent appelés *Legacy*, ou les systèmes extérieurs à l'entreprise. *Le Context Mapping* consiste à **cartographier** les contextes bornés du modèle dans leur écosystème et à caractériser leurs relations.

Les contextes échangent entre eux des données. Ces flux de données servent de base pour schématiser les canaux de communication. Quand on effectue le *Context Mapping*, on ne s'intéresse pas aux données en détail, mais on se place à un niveau plus haut, pour considérer en premier lieu la **sémantique** et l'**organisationnel**, puis les aspects liés aux flux de données, comme le **transactionnel** et la **synchronisation**. Nous allons voir ces différents points et détailler quelques grandes stratégies pour faire communiquer les contextes entre eux.

Sémantique

Chaque contexte a son langage spécifique. Le contexte borné est défini avec son langage commun. Un système *Legacy* comporte également un ensemble de termes et de concepts qui forment un langage, même si moins cadré : terminologie moins rigoureuse et plus ambiguë, mélange de plusieurs domaines fonctionnels, ou mélange de fonctionnel et de technique.

Lorsque les frontières entre contextes sont floues ou poreuses, les contextes ont tendance à déborder et fuir les uns dans les autres, et les langages à se mélanger. L'objectif est de définir les frontières autour de chaque contexte et de les rendre explicites. Pour éviter toute pollution entre langages, toute corruption de modèles entre eux et ainsi respecter le principe de *Separation of Concerns*, il faut limiter les données échangées et définir des passerelles, des traductions entre concepts.

Organisationnel

Les frontières au sein d'un système d'informations, c'est-à-dire son architecture, sont soumises à une forte contrainte en relation avec l'organisation dans l'entreprise. Ce constat a été formalisé sous le nom de **loi de Conway**, reformulée ainsi :

> *« If the architecture of the organization is at odds with the architecture of the system, the architecture of the organization wins. »*

Ruth Malan[33]

Même lorsque les frontières sont claires pour tout le monde, les relations entre contextes peuvent imposer certaines contraintes dont il est difficile de se rendre compte au moment du design de ces relations. De telles contraintes peuvent se faire ressentir dans le rythme de livraison d'un produit.

D'une manière générale, on constate trois types de relations entre contextes :

1. *Upstream/downstream*
2. *Mutually dependent*
3. *Free*

33. Ruth Malan, Software Architect

Upstream/downstream

L'expression *upstream/downstream* est empruntée au vocabulaire des flux mais ne porte pas sur la direction des données. Elle a trait à divers aspects : organisationnels (équipes, planning…) ou techniques (code, performance, réactivité…), ce qui est également valable pour les deux autres types de relations, *mutually dependent* et *free*.

Une relation *upstream/downstream* est caractérisée par le fait que les actions du groupe en amont (*upstream*) affectent le groupe en aval (downstream) sans que la réciproque soit vraie : les actions du groupe en aval n'affectent pas de manière significative les projets en amont. On peut faire l'analogie avec deux villes le long d'un même fleuve (Client/Fournisseur) : la pollution générée par la ville en amont affecte nécessairement la ville en aval.

Mutually dependent

Une situation plus délicate que l'on aimerait **éviter** est celle du *mutually dependent*. C'est le cas lorsque deux produits, chacun dans son propre contexte, doivent être livrés en même temps pour que la livraison réussisse. Les deux produits sont interdépendants, l'un nécessitant des données ou une fonctionnalité de l'autre. C'est d'autant plus embêtant lorsque le produit apportant le moins de valeur est celui le plus compliqué à livrer et dont la livraison est la plus susceptible d'échouer, entraînant l'échec du produit de plus grande valeur. Les deux équipes vont alors se retrouver en situation de dépendance mutuelle, obligées de livrer en même temps. Le couplage entre les deux systèmes est fort.

Free

L'idéal est celui d'un contexte *free*, dont le succès de la livraison n'est pas affecté par les échecs des contextes dont il dépend.

Permettre une livraison par parties (par exemple avec une architecture en micro-services[34]), supporter plusieurs versions d'API[35] en parallèle, dissocier livraison et activation de fonctionnalités au moyen de *feature toggles*[36] nous aident à nous rapprocher d'un contexte *free*.

34. https://fr.wikipedia.org/wiki/Microservices
35. https://restfulapi.net/versioning/
36. https://martinfowler.com/articles/feature-toggles.html

Context Patterns

Voyons désormais des patterns communs permettant de caractériser et de contrôler les relations entre contextes.

Partnership

On parle de partenariat lorsque deux équipes travaillant sur des **contextes distincts** réussissent ou échouent ensemble. Les deux équipes ont reconnu qu'elles étaient *mutually dependent* et travaillent désormais en **collaboration étroite**, coordonnant leur planning de développement et de déploiement, gérant de manière conjointe l'intégration de leurs produits.

La plupart du temps, il n'est pas nécessaire que les développeurs comprennent en détail le modèle de l'autre sous-système. En revanche, leur planning étant intriqué, dès que le développement dans une équipe rencontre un sérieux problème, ils doivent trouver ensemble une solution rapide pour débloquer la situation. Une telle solution peut entraîner une **dette technique** alors choisie par les deux équipes, qui devront de nouveau synchroniser leur planning pour laisser le temps à l'équipe touchée par la dette de la combler avant que les intérêts ne se soient trop accumulés.

Pour pérenniser l'intégration de leurs produits, on peut mettre en place une suite de tests spécifique qui apporte la preuve de leur bon interfaçage. Ces tests peuvent être automatisés en les incluant au processus d'intégration continue *(CI)*. On parle ainsi de *Contract Testing* et de *Consumer Driven Contracts*, le client délivrant au fournisseur le **contrat** résultant du « pacte » mis en place entre eux, charge au fournisseur de vérifier ce contrat avant tout déploiement d'une nouvelle release. PACT[37] est un outil multiplateforme spécialisé dans ce type de tests.

Shared Kernel

Cette situation se produit lorsqu'**une partie du modèle** et du code associé est partagée avec un autre contexte pour former un noyau commun *(Shared Kernel)*. Les deux contextes sont alors en interdépendance très étroite, ce qui peut autant faciliter que miner le travail de conception.

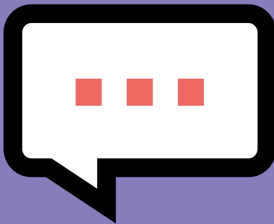
Fil rouge

Dans le cadre de notre projet de recrutement chez SOAT, la stratégie de sécurité ou l'authentification peuvent être mutualisées entre plusieurs contextes bornés et ainsi former un *Shared Kernel*.

Cela part d'une bonne intention : éviter la duplication de code et surtout de concepts, c'est-à-dire suivre le principe *DRY (Don't Repeat Yourself)*. Cependant, cela va créer un **couplage fort** entre les contextes mutualisant le noyau commun. Cela va complexifier l'intégration des systèmes sous-jacents et augmenter le besoin en coordination. Dans ce cas il ne faudrait qu'**une seule équipe** parmi les équipes concernées, qui s'occuperait de ce contexte partagé.

37. PACT : https://docs.pact.io/

De ce fait, il vaut mieux privilégier un noyau commun de **petite taille**, avec des frontières explicites, par exemple un *repository git* distinct pour le code, et un processus d'intégration continue. Cela nécessite absolument qualité, fréquence de communication et collaboration entre les équipes partageant un noyau commun. Les équipes doivent être **matures.**



En pratique, il est souvent plus simple de dupliquer dans chaque contexte les éléments actuellement partagés. Il s'agit en fait de **duplication accidentelle** : comme ces éléments sont désormais dans des contextes bien distincts, ils vont évoluer indépendamment, selon des besoins différents, ce qui va dissiper ce qui ressemblait à de la duplication.

Customer/Supplier Development

La situation de base est celle où deux équipes sont dans une relation *upstream/downstream*.

Cela peut correspondre à une diversité de situations comme par exemple :

- L'équipe en aval peut être impuissante et à la merci des priorités de l'équipe en amont.
- Elle peut être ralentie par une lourdeur administrative dans les demandes de changement ou par un processus complexe d'approbation.
- La situation en amont n'est pas forcément plus facile. L'équipe en amont peut être inhibée de crainte de briser les systèmes en aval. Ses développements peuvent être stoppés si l'équipe en aval peut exercer un droit de veto.

La solution peut consister à établir une **relation de client/fournisseur** entre les deux équipes. Cela signifie que les **priorités** de l'équipe en aval sont prises en compte par l'équipe en amont. Les équipes **négocient** budget et planning et comprennent les engagements respectifs qui en découlent.

L'équipe en aval joue alors le rôle de client. Dans un contexte agile, elle peut participer aux sessions de planification de l'équipe en amont portant sur leurs sujets en commun. Les deux équipes collaborent à la définition des **critères d'acceptation** et à l'élaboration des tests d'acceptation associés, qui permettent de valider le contrat client (que doit fournir l'équipe en amont). Ces tests peuvent être automatisés en les incluant dans la chaîne d'intégration continue *(CI)* de l'équipe en amont. Cela permet à cette dernière d'apporter des changements sans crainte d'effets secondaires en aval. Cette suite de tests peut être formalisée dans le cadre d'un *Contract Testing* déjà évoqué pour le *Shared Kernel.*

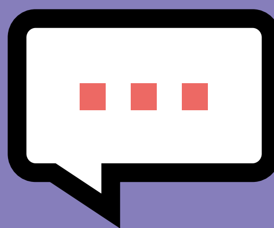
Conformist

Ce pattern se produit lorsque deux équipes sont en relation *upstream/downstream* mais, contrairement au pattern *Customer/Supplier*, l'équipe en amont n'a aucune motivation ou aucun moyen de satisfaire les besoins de l'équipe en aval. L'équipe en aval n'a **aucun contrôle** sur cette situation.

Elle se retrouve dans l'attente de fonctionnalités qui ne viendront jamais. Le mieux pour l'équipe en aval est de comprendre qu'elle ne disposera de rien de plus de la part de l'équipe en amont. Elle doit **se conformer** à ce qui est en amont.

Fil rouge

Dans le cadre du recrutement chez SOAT, le contexte borné *Communication* dépend de la suite Google, qui est une solution d'entreprise assez générique. SOAT étant une PME, elle n'a pas les moyens de négocier des services adaptés spécifiquement à ses besoins. Ce contexte borné ne correspond pas à un sous-domaine *Core*. Il peut donc être de type *Conformist*. Cela simplifie grandement la situation, car il n'y a pas à se préoccuper de traductions entre les deux contextes. Le contexte en aval partage le langage commun du contexte en amont.



Dans le cas d'un sous-domaine *Core*, il vaudrait mieux s'orienter vers une couche anti-corruption *(ACL)*, plus coûteuse mais alors nécessaire pour garantir l'intégrité de son modèle.

Anticorruption Layer (ACL)

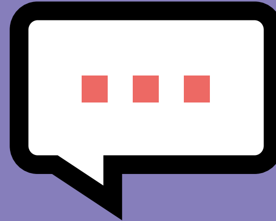
Nous avons déjà évoqué le besoin de traduction entre deux contextes. Lorsque ces contextes sont bien bornés et bien conçus, que la collaboration entre les équipes marche bien, les couches de traduction sont des solutions simples et parfois élégantes. Mais lorsque le contrôle ou la communication ne suffisent pas à établir une relation en *Shared Kernel*, *Partnership* ou *Customer/Supplier*, la **traduction** devient plus complexe. La couche de traduction prend une tournure plus défensive, et se mue en zone démilitarisée.

Cela intervient dans ce type de situations. Face à un système en amont fournissant une interface pléthorique, le système en aval risque de ne pas faire le poids et de corrompre son modèle pour ressembler au modèle en amont. Ces mastodontes peuvent en de rares exceptions être bien conçus mais cela peut ne pas convenir au modèle en aval qui ne peut se comporter en *Conformist*. Mais dans la majorité des cas, il s'agit de systèmes ***Legacy*** avec un modèle défaillant et ressemblant plutôt à une *Big Ball of Mud*.

La solution proposée avec l'ACL repose sur une protection active pour limiter les impacts des changements du système en amont. La couche d'anti-corruption joue le rôle de filtre protecteur. Tout élément extérieur au contexte borné ne peut y entrer qu'en passant par l'ACL qui en sélectionne les constituants requis (et permis), et les traduit dans le **langage commun du contexte borné** en aval.

Peu ou pas de modifications du système en amont sont requises car l'ACL préserve l'interface existante du système en amont. L'ACL s'occupe de faire la traduction de haut en bas ou dans les deux sens entre les modèles. L'ACL effectue également un filtre pour enlever les concepts qui n'ont pas d'équivalent, et ne fournir que les données nécessaires en aval.

On évite ainsi une **pollution du système en aval.** Cela permet également de respecter le principe de ségrégation des interfaces[38].

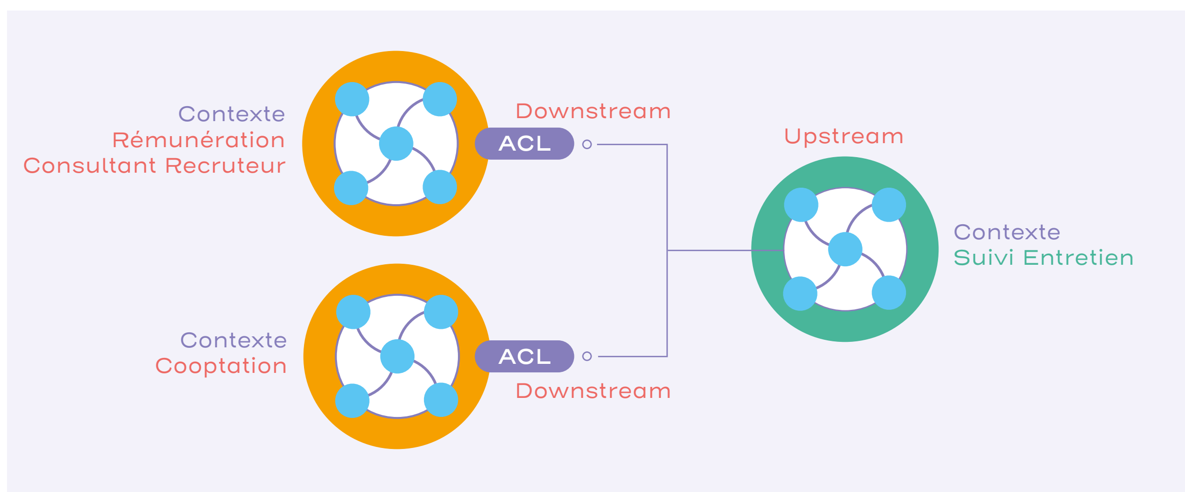


Dans certains cas, l'ACL peut aller plus loin et contenir des instructions techniques en plus de sa responsabilité de traduction.

Open Host Service

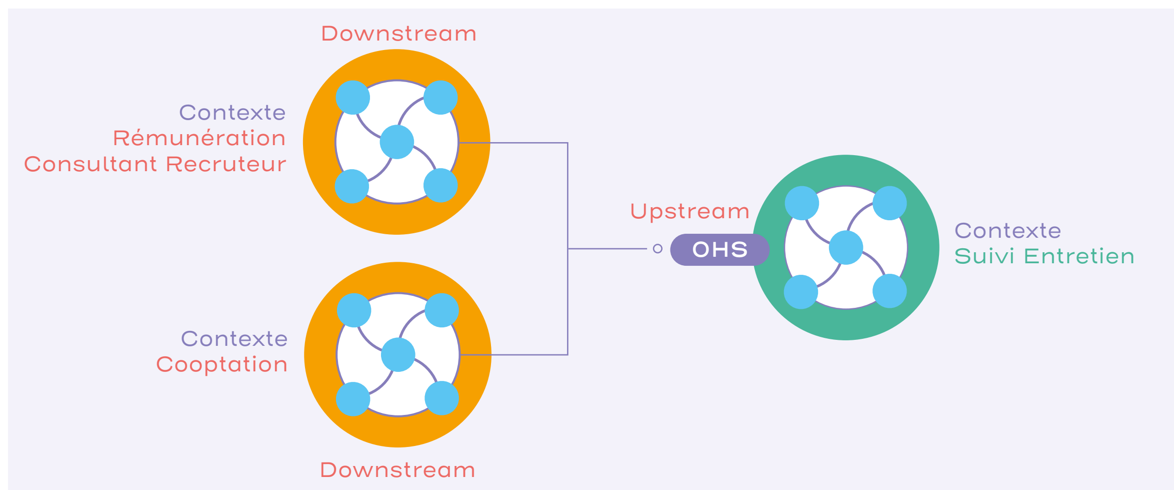
Pour un besoin ponctuel, l'approche ACL permet d'éviter la corruption des modèles avec un **coût minimum.** Dans le cas d'un *upstream* (fournisseur) ayant plusieurs *downstreams* (clients), cette personnalisation peut être trop lourde à gérer côté maintenance, et tout changement pourrait avoir des conséquences pas toujours maîtrisables. C'est un cas où la complexité dépasse le seuil des capacités de l'équipe.

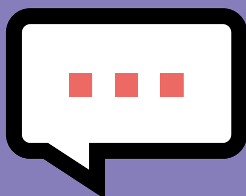
Une solution plus générique et offrant plus de flexibilité peut être envisagée.



38. https://en.wikipedia.org/wiki/Interface_segregation_principle

Le contexte en amont peut se conformer aux besoins des contextes en aval sous la forme d'un ensemble de services. De nouveaux services peuvent être ajoutés à la demande, tant que cela reste dans un cadre générique pour éviter les problèmes de personnalisations complexes.



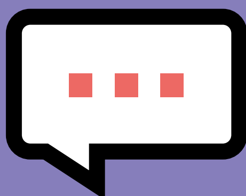


Un contexte peut être *Open-Host Service* pour certains contextes client seulement. Il redevient un contexte lambda pour les autres, pouvant établir avec eux n'importe quel type de relation.

Published Language

La traduction directe entre des modèles existants n'est pas toujours une bonne solution. Ces modèles peuvent être trop complexes ou mal conçus, leur documentation absente ou parcellaire. Si le langage de l'un des contextes est utilisé comme langage d'échange de données, cela va dans l'ensemble le figer et il ne pourra que **difficilement évoluer** pour répondre à de nouveaux besoins de développement.

La solution consiste alors à mettre en place un langage commun d'échange entre les deux contextes. Le langage sera suffisamment bien documenté pour permettre son utilisation sans encombre, et **facilitera les traductions.**



De nombreuses industries des secteurs bancaires et aériens établissent des *Published Languages* sous la forme de normes d'échange de données. Les équipes projet élaborent également leurs propres *Published Languages* qui seront utilisés au sein de leur organisation[39].

39. Le *Published Language* est souvent combiné avec un *Open Host Service*

Separate Ways

L'objectif est de séparer deux contextes faiblement liés entre eux. Cela permet de se focaliser sur un plus petit périmètre fonctionnel et de trouver des solutions simples qui lui sont spécifiques. En effet, l'intégration de deux sous-systèmes est toujours coûteuse alors que les avantages sont parfois minimes, comme nous l'avons constaté dans le pattern *Partnership*.

On procède en examinant les besoins métier. S'il s'avère que deux ensembles de fonctionnalités n'ont pas de relation significative, ils peuvent être complètement séparés l'un de l'autre.

Une autre situation pouvant bénéficier des *Separate Ways* est celle où deux équipes travaillent avec un **décalage horaire trop important**. La collaboration entre les deux équipes est alors quasiment impossible. Si le domaine fonctionnel peut être découpé en deux, cela permet aux équipes d'être libres et de **soulager l'organisation**. Au niveau du code, cela se traduira par un peu de **duplication** mais qui devrait s'avérer accidentelle.

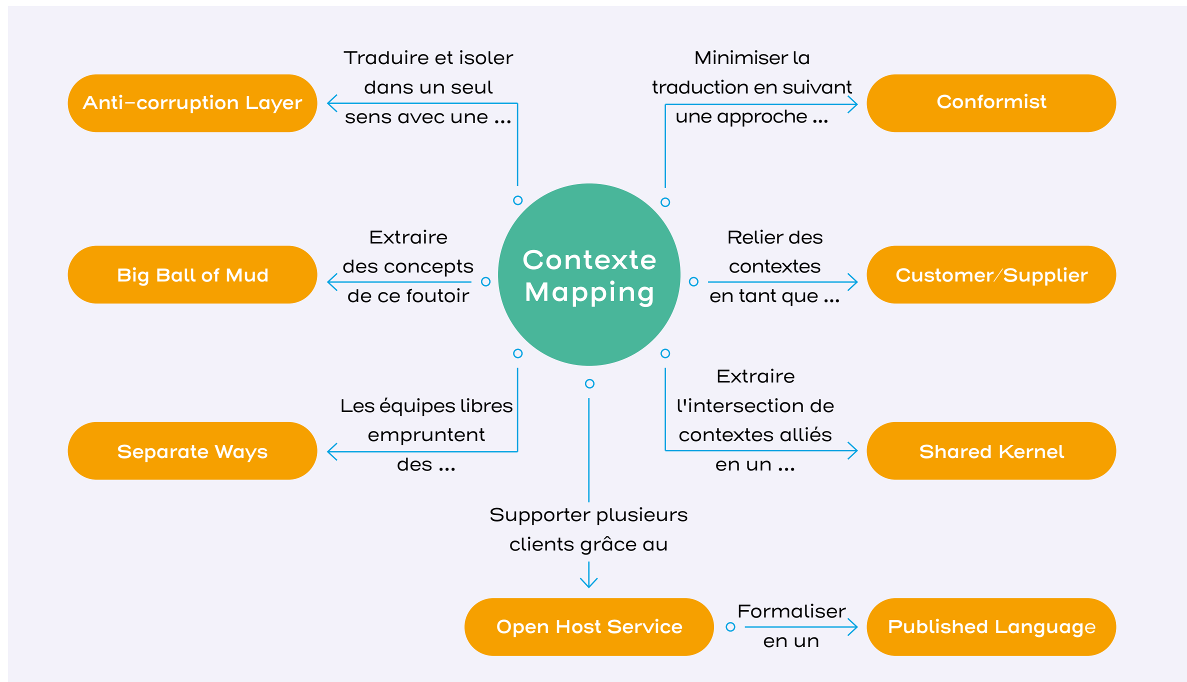
Big Ball of Mud

Cette situation correspond à un vaste système monolithique et ***Legacy***, portant sur plusieurs modèles larges et entremêlés. Ils ont d'ailleurs tendance à se répandre dans les systèmes voisins.

Autant ces systèmes apportent encore de la valeur en restant générateurs de revenus pour l'entreprise, autant interagir avec eux est délicat. Il ne faut pas chercher à les améliorer en leur appliquant les méthodes DDD. C'est peine perdue. Dans la cartographie des contextes, le mieux est de les **délimiter dans un seul paquet**, par exemple affublé d'une tête de mort ou d'une bombe, ou désigné en tant que *Big Ball of Mud*[40].

Synthèse

Voici comment Eric Evans schématise les patterns du *context mapping* :

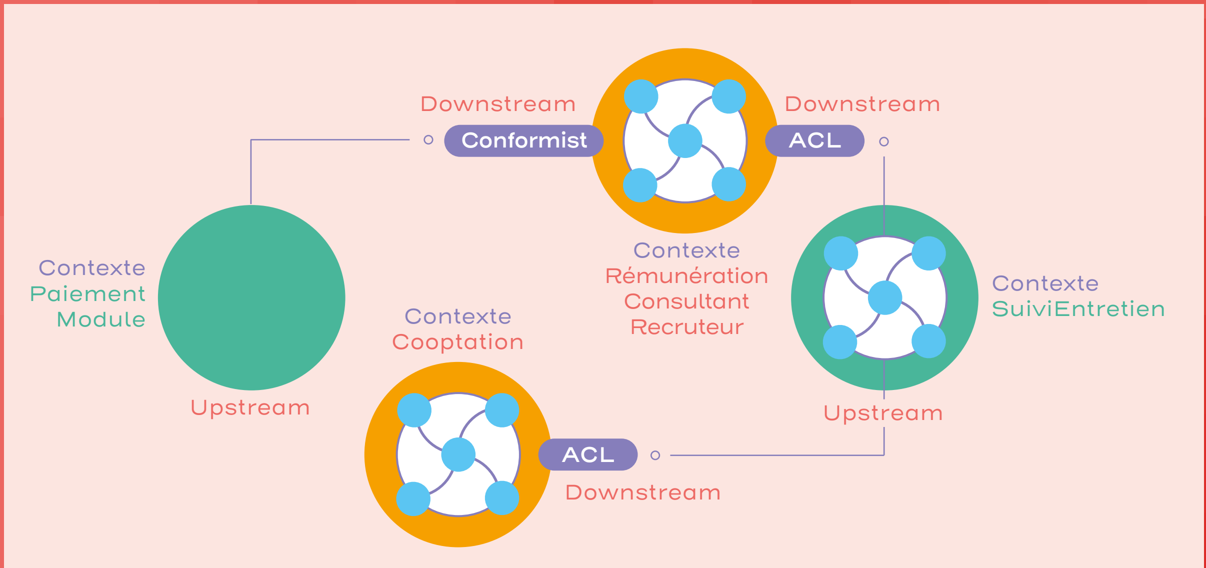


40. Plus d'informations sur la BBoM ici : http://www.laputan.org/mud/mud.html

Fil rouge

Voici la cartographie d'une application de recrutement faisant intervenir 4 contextes, dont l'un est en aval avec une relation conformiste d'un côté et protégée par une couche anti–corruption de l'autre.

Dans ce modèle, le contexte "Rémunération" doit se conformer au contexte de "Paiement" lors des changements. De l'autre côté, nous aimerions protéger ce contexte "Rémunération" des impacts éventuels d'un système Legacy qui est celui de "Suivi des entretiens".



Conseils généraux

La cartographie doit représenter l'**état actuel** du système global, et non son état souhaité pour le futur. Si c'est le bazar complet, indiquez-le. Les relations entre équipes ne sont pas toujours connues ou nettes. On peut aussi l'indiquer. Cela permet d'attirer l'attention sur les problèmes et les risques associés au projet.

Voici comment l'on peut procéder pour cartographier les contextes :

1. Dessiner chaque contexte et indiquer son nom issu du langage commun s'il existe ou s'il est à construire
2. Définir quels modèles communiquent entre eux et évaluer leur « bande passante »
3. Indiquer la direction d'une relation (*upstream/downstream*) quand cela s'applique
4. Mettre le nom du pattern sur la relation : *Partnership, Shared Kernel, Customer/Supplier, Conformist, Separate Ways...*
5. Signaler les contextes *Big Ball of Mud*
6. Désigner les contextes bornés qui pratiquent traduction ou partage : *ACL, Open-host service, Published Language*

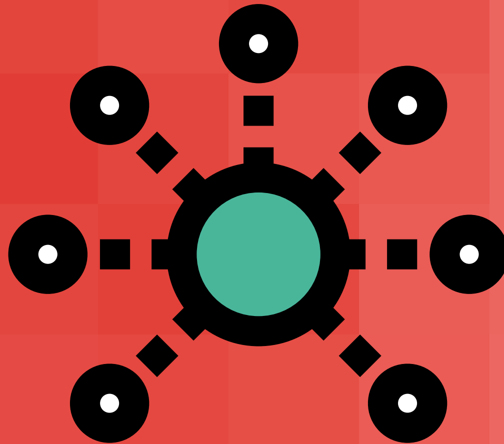
Commencer du DDD entouré de systèmes *Legacy* n'est pas évident. L'utilisation explicite d'un contexte borné et de patterns stratégiques peut permettre de greffer de nouvelles fonctionnalités dans une telle situation, sans chercher à refaire entièrement les systèmes *Legacy*.

Les Bounded Contexts en résumé

Eric Evans décrit des stratégies[41] selon l'échelle d'application du DDD. Voici un résumé des trois premières :

1. ***Bubble context*** : c'est la forme la moins ambitieuse mais la plus adaptée quand on manque d'expérience en DDD. Il s'agit de créer un nouveau contexte qui va s'appuyer sur quelques données du *Legacy* et protéger son modèle avec une *ACL*. Le contexte est comme une bulle de savon : il aura naturellement tendance à se faire réabsorber par le *Legacy*. Pour autant, les nouvelles fonctionnalités auront pu être développées et continueront de fonctionner comme tout bon système *Legacy*.
2. ***Autonomous bubble*** : il se différencie du *Bubble Context* par sa propre base de données. Cela facilite l'ajout de nouvelles données mais requiert un travail sur l'ACL pour ajouter la synchronisation des données communes avec le *Legacy*.
3. ***Legacy assets as services*** : il s'agit d'appliquer le pattern *Open-Host Service* pour faciliter l'interfaçage du *Legacy* avec de nouveaux contextes. Cela permet un investissement plus judicieux en étant plus important côté *Core* que *Generic/Supportive* où se situe généralement le *Legacy*.

41. http://domainlanguage.com/wp-content/uploads/2016/04/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf



03
Tactical design

Cette partie concerne les briques du *model-driven design*. Il s'agit de patterns de conception orientée-objet (OO) qui forment un tout cohérent, un ensemble de bonnes pratiques recommandées pour concevoir et implémenter le modèle. Utilisés à bon escient, ils aident à établir et maintenir un modèle de code « souple » *(supple design)* et à isoler le domaine pour ne pas le polluer de considérations techniques *(separation of concerns)* et ainsi éviter de la complexité accidentelle.

Les deux autres piliers du DDD sont essentiels pour la partie tactique. Il s'agit de découvrir non seulement *ce que l'on doit coder* (*what*) mais aussi *pourquoi on le fait* (*why*) et combien d'efforts investir. Ces préoccupations sont des facteurs plus forts de réussite que celles concernant la *façon de les implémenter* dans le code (*how*). Le code n'est qu'un produit dérivé de tout le travail en amont de modélisation collaborative et de recherche de la stratégie la plus efficace.

Tactical design et Programmation Fonctionnelle

Un développement suivant le paradigme orienté-objet gagnera à utiliser ces patterns, de manière plus ou moins poussée selon qu'il s'agisse du système *Core* ou pas. Mais la réciproque est fausse : DDD est « techno-paradigmo-agnostique ».

Le développement d'un système *Core* peut par exemple tout à fait être réalisé en programmation fonctionnelle[42]. Il faut donc garder en tête que les patterns tactiques ne sont alors pas forcément les plus adaptés et les plus efficaces.

Les grands principes

Avant d'étudier chaque pattern, voyons-en les grands principes :

- L'objectif est d'éviter un *modèle de domaine anémique*, c'est-à-dire des objets métier sans comportement et contenant uniquement des données à ne pas confondre avec les *DTO* (Data Transfer Object) qui contiennent uniquement des données.
- Le modèle est exprimé au moyen d'objets de type *Entity* (Entité) et *Value-Object*, puis *Service* et éventuellement *Domain Events* (événements du domaine).
- L'encapsulation est renforcée grâce aux *Aggregates* (Agrégats).

42. Cf. livre déjà cité *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*

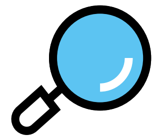
- Les événements du domaine permettent de notifier des changements d'état dans des entités.
- Les sources de données comme la persistance ou une API externe sont abstraites, au moyen de *Repositories* associés à des agrégats.
- Le domaine est isolé du reste grâce à une architecture en couches, centrée sur le domaine.

Identité et cycle de vie

Entity et Value-Object sont les deux patterns tactiques clés. Ces deux types d'objets se distinguent sur les notions d'identité et de cycles de vie.

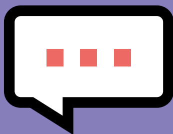
Certains objets sont intrinsèquement définis par la **continuité** de leur **identité** et non par leurs attributs (qui peuvent, eux, varier dans le temps). Ces objets ont une identité qui leur est propre : elle est **unique**, ce qui permet de distinguer les instances entre elles à l'instant T. Leur identité est **intangible** : elle ne varie pas dans le temps et les suit tout au long de leur cycle de vie.

D'autres objets n'ont pas de cycle de vie propre. Ils sont définis uniquement par leurs attributs. Cette distinction entre objets avec ou sans identité est propre au modèle en cours, dans son contexte *(Bounded Context)*.



Exemple

Prenons l'exemple d'un billet de banque. Chaque billet est porteur d'un numéro unique qui l'identifie. Si deux billets ont le même numéro, l'un est un faux. Cependant, selon le contexte de notre problème, on peut prendre en compte cette identité (comme pour un système de la Banque de France) ou l'ignorer (comme dans le cas d'un distributeur de billets, pour qui seules la devise et la valeur faciale comptent).



L'identité est un vrai concept métier. Ce n'est pas qu'un simple artefact technique, tel un GUID ou une colonne *identity* qui peut servir de clé primaire en base de données.

En revanche, il faut éviter de prendre un attribut naturel (qui vient du monde réel) comme identifiant car il peut changer : le nom d'une personne, son adresse email ne sont ni **immuables**, ni **uniques**.

C'est pourquoi une **clé primaire** ou un **identifiant métier** représente une bonne approximation du concept d'identité en DDD. On peut ainsi utiliser l'identifiant d'une Entité pour la désigner[43].

43. Par exemple dans un événement qui s'y rapporte, plutôt que de fournir l'objet entier représentant l'*Entité*

Entité

Une *Entité* est un objet pourvu d'une **identité** et d'un **cycle de vie.**

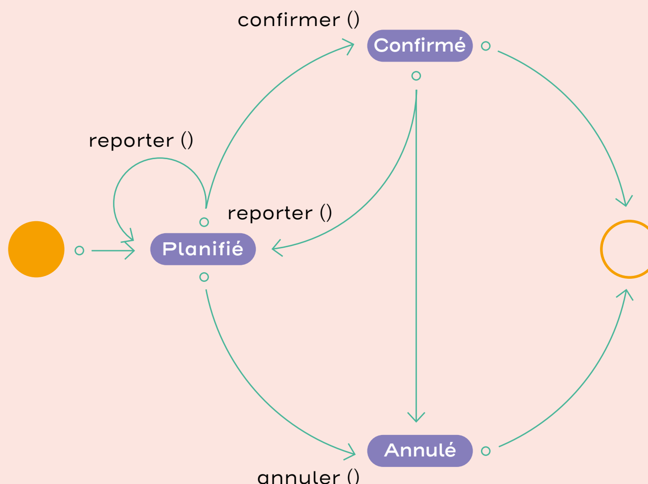
Nous pouvons faire un parallèle entre l'**identité** et le numéro de suivi d'un colis. Si l'on devait modéliser le système d'un transporteur où chaque colis est suivi individuellement, le Colis serait une *Entité* dont l'identité est représenté par son numéro de suivi *(tracking)*. Cet identifiant permet à l'expéditeur et au destinataire de suivre le colis tout au long de son cycle de vie, en renseignant ce numéro dans une application ou un site web :

Gestion de l'identité pour une entité *Colis*

Fil rouge

Dans le cas de notre fil rouge, un *Entretien* possède un cycle de vie et une identité propre, qui ne varie pas dans le temps : quand on parle de l'entretien de M. X ou «entretien n°12345» en nous servant d'un numéro unique, cela suffit à identifier cet entretien sans avoir à prendre en compte l'instant et son état. L'*Entretien* est donc une *Entité*.

Cycle de vie du statut de l'objet *Entretien* : 3 états possibles



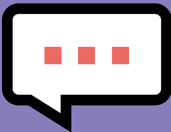
Ne pas confondre l'*Entité* au sens DDD avec une entité d'un **ORM** *(Object Relation Mapping)* tel qu'*Entity Framework* ou *Hibernate*. Il s'agit de concepts différents mais aussi de considérations différentes : le métier d'un côté, la persistance de l'autre. Même s'il s'agit d'objets avec des attributs identiques, dans l'idéal, ils seront distincts et placés dans des couches différentes, *Domaine* et *Infrastructure*[44].

Value-Object

Caractéristiques d'un *Value-Object*

Un *Value-Object* est un objet contenant un état mais **dénué d'identité.** Il est entièrement défini par ses attributs et il est **dépourvu de cycle de vie.**

L'objet est conceptuellement similaire à une valeur d'un type primitif, d'où son nom. Cet objet est **immuable** afin de se prémunir contre les effets de bord liés à la mutation d'états, en particulier lorsqu'il est **partagé.**



Les *Value-Objects* doivent représenter des concepts du domaine métier.

Un tel objet aura ses **attributs en lecture seule**, et il seront définis uniquement lors de sa construction. On cherchera à rendre le maximum d'attributs privés afin de privilégier l'encapsulation des **comportements** dans l'objet, et le principe *Tell, don't ask*[45]. L'égalité se fait par valeurs, en comparant attribut par attribut. Une action aboutissant à un changement d'état d'un *Value-Object* se fait en renvoyant une nouvelle instance plutôt que par une mutation de l'instance en cours.

---

44. Cf. article *Having the domain model separated from the persistence model*, https://enterprisecraftsmanship.com/2016/04/05/having-the-domain-model-separate-from-the-persistence-model/
45. https://www.martinfowler.com/bliki/TellDontAsk.html

```
class Salle {
  constructor(
    readonly nom: string,
    readonly nombreDePlaces: number,
  ) {}

  equals(other: Salle) {
    return this.nom === other.nom
        && this.nombreDePlaces === other.nombreDePlaces;
  }

  reserver(creneau: Creneau) { /*...*/ }
}
```

> « *When you care only about the attributes and logic of an element of the model, classify it as a* ***Value-Object.*** »
>
> Eric Evans

Les *Value-Objects* sont également **interchangeables.**

Fil rouge

Pour effectuer un entretien d'embauche, une salle peut être remplacée par une autre salle de capacité compatible (c'est son attribut *Capacité* qui est important ici). Dans ce contexte, la *Salle* est un *Value-Object*. Pour la réservation d'une salle, on passe par un outil externe (sous-domaine *supportive*) dans lequel la salle est identifiée de manière unique, ce qui en fait une *Entité* dans ce contexte extérieur. Les deux ne sont pas incompatibles.

La relation avec *Entité*

En corollaire de l'absence d'identité et de cycle de vie, dans la plupart des cas, les *Value-Objects* ne peuvent pas exister sans qu'une *Entité* parent n'en soit propriétaire, dans une relation de composition entre objets[46]. Le *Value-Object* est un attribut de son *Entité* parent avec non pas un type primitif mais un type créé pour nos propres besoins.

Persistance

Autre considération pratique : lorsqu'il s'agit de **persister en base de données** un *Value-Object*, nous vous conseillons de le mettre dans la même table que celle de son *Entité* parent. C'est seulement lorsque l'*Entité* est une vraie collection de *Value-Objects* sans limite de taille qu'il faut considérer une table spécifique pour stocker l'association.

Ce qui veut dire que c'est une sorte de grappe d'objets qui sont fortements liés du point de vue métier (Agrégat) que l'on persiste.

Identification d'un *Value-Object*

46. En pratique, un Value Object peut exister sans une Entity

Comment savoir qu'un concept de domaine mériterait d'avoir son propre *Value-Object* ? Pour cela, nous pouvons évaluer les critères suivants, proposés par Vladimir Khorikov[47] :

- Le nombre et la complexité des invariants
- Le nombre d'attributs primitifs qu'il embarque
- La complexité globale du domaine
- Le nombre de duplications évitées

Le nombre et la complexité des invariants du domaine. Chaque concept du domaine est corrélé à un certain nombre d'***invariants***, c'est-à-dire des règles valables dans certains cas d'utilisation et qui sont maintenues même si les données changent. Plus ces règles sont nombreuses, plus il est préférable de les intégrer dans leur propre objet. Exemples : un *Email* comporte plusieurs règles de validation assez complexes. Un *NombrePositif* n'en a qu'une seule et le concept est assez simple. Seul l'*Email* se prête bien à la modélisation en *Value-Object*.

Le nombre d'attributs primitifs qu'il embarque. Chacun des exemples précédents, *Email* et *NombrePositif*, ne comportait qu'un attribut primitif. Il est cependant fréquent d'avoir au moins deux attributs fortement liés entre eux, tels qu'un objet *Money* contenant une valeur faciale et une devise. Dans ce cas-là, nous vous conseillons d'avoir recours au *Value-Object*.

La complexité globale du domaine. Comme nous l'avons vu pour le DDD en général, introduire un *Value-Object* représente un investissement qui paie si le domaine est suffisamment complexe.

Le nombre de duplications évitées. Plus un concept dans un contexte particulier *(Bounded Context)* est utilisé, mieux il vaut encapsuler ses invariants dans un objet spécifique. On évite ainsi de répéter la vérification de ces invariants. A contrario, s'il n'y a qu'un seul usage, on peut l'isoler dans une méthode sans en faire une classe à part.

Attention à ne pas voir des *Value-Objects* partout. Ce n'est pas qu'une question de *Primitive Obsession*[48] ou inversement d'*over-design*. Ne laissez pas les problèmes techniques s'insinuer dans les parties centrales de votre modèle de domaine.
Un objet *Optional70LimitedString* répond à une préoccupation purement technique.
S'il existe un vrai concept métier, donnez lui un nom métier, et pas technique.

Intérêts des *Value-Object*

Les *Value-Objects* sont simples à utiliser : ils peuvent être facilement créés ou supprimés car il n'est pas nécessaire de maintenir une identité. Ils forment une unité logique et insécable. Ils contiennent leurs règles de validation spécifiques, ce qui permet de manipuler des objets valides plutôt que de vérifier avant chaque opération s'ils le sont.

On passe d'une programmation défensive (travail en aval) à un design défensif (pré-travail en amont), plus élégant et plus simple à utiliser.

Cette **simplicité d'utilisation** en fait les candidats idéaux pour y mettre les règles métier telles que les *invariants*.

47. https://enterprisecraftsmanship.com/2017/06/15/value-objects-when-to-create-one/
48. https://refactoring.guru/smells/primitive-obsession

Fil rouge

Dans le cas de notre fil rouge, la réservation d'une salle ne peut s'effectuer que pendant les horaires d'ouverture des locaux de SOAT : 08h00-20h00. Cette règle est de la responsabilité du *Value-Object* ***Salle*** et devrait donc être encapsulée dans son code.

Value-Object vs DTO

Un *DTO, Data Transfer Object*, est un objet regroupant un ensemble de données. Il est utilisé pour **transférer des données** entre couches applicatives ou entre systèmes. Pour faciliter ce transfert, les données peuvent être aplaties plutôt que sous forme hiérarchique plus cohérente. De même, l'objet peut être sérialisable. Il peut ainsi comporter des *getters* et *setters*. Dans tous les cas, il n'a conceptuellement pas besoin d'être immuable.

Un *DTO* ne contient aucune logique, aucun comportement. Un *DTO* ne peut donc pas être un *Value-Object*, ce dernier ayant pour rôle d'encapsuler de la logique métier au sein d'une structure cohérente.

	DATA TRANSFER OBJECT (DTO)	VALUE OBJECT (VO)
Intention	Transfert de données	Représentation d'un concept du Domain Model
Contenu	Ensemble des données à faire transiter (réseau, passage entre couches...) Peut agréger des données hétérogènes pour un cas d'utilisation	Des données fortement cohérentes
Comportement	Pas de comportement	**Comportement riche**
Immuable	Généralement non	Oui

Value-Object vs Value Type

Ne pas confondre *Value-Objects* et *Value Types* .NET, les «*structs*». Les *Value-Objects* sont des concepts du domaine tandis que *structs* sont un détail d'implémentation sur la manière dont les objets sont stockés en mémoire.

On peut les confondre car les *structs* sont immuables et passés par valeur. L'objet *DateTime* est une *struct* de la bibliothèque standard .NET qui n'a pas d'identité propre : il a donc la sémantique d'un *Value–Object*. Mais c'est un objet répondant à des problématiques techniques.

En théorie, on pourrait implémenter un *Value–Object* sous la forme d'un *struct*. Cependant, nous ne vous le conseillons pas : l'absence d'héritage oblige à coder à la main l'égalité par valeur et empêche nombre d'ORM de mapper vers des *structs*.

Entité ou Value–Object

Entité et *Value–Object* se différencient sur les critères suivants :

	VALUE-OBJECT (VO)	ENTITY
Identité intrinsèque	Non	Oui
Egalité	Égalité structurelle, sur un ensemble métier d'attributs	Egalité des identités *(parfois égalité par référence)*
Comportement lors d'un changement d'état	Instance immuable → Production d'une nouvelle instance	→ Mutation de l'instance courante
Comportement temporel	Intemporel	Variation de l'état dans le temps
Autonomie *(vis-à-vis d'un cas d'utilisation)*	Ne peut pas être utilisé seul Caractérise une Entité parent	Peut être utilisée seule
Composition : *peut agréger*	Autres *Value-Objects*	*Value-Objects* et autres *Entités*

Le choix entre une *Entité* et un *Value–Object* **dépend du contexte.** Prenons l'exemple des billets de banque. Pour monsieur tout le monde, deux billets de 50 € sont interchangeables tant qu'ils sont vrais. Un *Value–Object* conviendrait. Pour les banques émettrices de billets, chaque billet est identifié de manière unique. C'est donc une *Entité*.

Si les connaissances actuelles du domaine ne permettent pas de trancher, partez sur un *Value-Object*, plus simple d'utilisation. Il sera toujours possible (même si pas forcément simple) de basculer ensuite sur une *Entité*.

En matière de code, il est important de pouvoir aisément distinguer un *Value-Object* d'une *Entité*. Une option commune que nous vous recommandons est d'utiliser soit des classes de base, soit un décorateur (@Entity ou @ValueObject), soit des classes ou interfaces génériques (`Entity<TId> et ValueObject<T>`)[49].

Agrégat

Définition d'un *Agrégat*

Un *Agrégat (Aggregate)* est un **groupe** d'objets considérés comme un **tout unique** lorsqu'il s'agit de modifier les données qu'il contient. Un *Agrégat* forme une **frontière** qui le sépare, lui et ses objets enfants, du reste des objets du modèle. Cette frontière émerge naturellement des cas d'utilisation et des ***invariants*** sous-jacents à respecter pour l'intégrité et la cohérence du groupe.

Fil rouge

Dans le cas de la planification d'un entretien technique chez SOAT, l'un des *invariants* est :
« Un *Entretien* est créé uniquement si le *ConsultantRecruteur* est disponible. »

Il est préférable de concevoir des petits *Agrégats* suivant les cas d'utilisation plutôt qu'un grand *Agrégat* général. Pour éviter certaines duplications dans un même contexte, on peut partager des *invariants* entre ces *Agrégats* au moyen de *Value-Objects* mis en commun.

Problématique

Les *Entités* du modèle ont une durée de vie, en mémoire, et sont généralement persistés en une base de données. La gestion de cette durée de vie n'est pas facile car il faut s'assurer de garder les *invariants* et la cohésion du modèle tout en autorisant les objets à avoir des relations entre eux, parfois soumises à contraintes.

Une solution consiste à raisonner avec un niveau d'abstraction plus haut, en agrégats d'objets plutôt qu'en objets pris séparément. Cela permet de réduire la complexité. Mais il faut souvent plusieurs modélisations successives avant de trouver l'*Entité*, existante ou émergente, qui est la mieux adaptée pour servir d'*Agrégat*.

Aggregate Root

Chaque *Agrégat* a un objet **racine** de type *Entité*[50]. L'*Aggregate Root* est le seul **point d'entrée** de l'*Agrégat* : il établit le lien entre les objets à l'intérieur et ceux à l'extérieur de l'*Agrégat*, grâce à l'**encapsulation et la portée des classes**.

Les données entrent et sortent de l'*Agrégat* sous la forme de *DTO* pouvant contenir des identifiants techniques d'*Entités*.

L'*Agrégat* est responsable de l'intégrité de ses données durant les **opérations** qui lui sont demandées.

49. Ces éléments peuvent contenir un code générique qui sert à tous les *ValueObjects* ou *Entities*
50. Dans des cas rares, peut être un *ValueObject*

Fil rouge

Dans notre fil rouge, les opérations de l'agrégat (qui sont uniquement des Commandes) *PlanificationEntretien* sont par exemple *Planifier, Confirmer, Reporter* et *Annuler.*

Domain Service (*Service*)

Caractéristiques d'un *Service*

- Une opération dans un *Service* fait référence à un concept du domaine qui n'appartient pas à une *Entité* ou à un *Value-Object*.
- Un *service* peut effectuer un traitement sur plusieurs *Entités/ Agrégats* ou *Value-Objects*.
- Les *services* n'ont pas d'état (***stateless***) : ils ne disposent pas de champs ni de propriétés.

Où mettre cette logique métier ?

Lorsque des comportements ne peuvent pas être naturellement associés à un objet, ls sont implémentés dans un *Service* du domaine (*Domain Service*).

Services et domaine anémique

Lorsque nos *Services* encapsulent tous les comportements métiers et que nos objets métiers (*Entity, ValueObject)* se retrouvent sans **comportement**, on parle de *Domaine Anémique*.

C'est l'inverse de la programmation orientée objet, dans laquelle on attend des objets ayant des comportements riches dans la couche *Domaine*.

Pour autant, les *services* ont leur place dans le modèle de domaine, afin d'accueillir les traitements qui ne trouvent pas une place naturelle dans les objets du domaine.

En programmation fonctionnelle, le problème ne se pose pas. Certes, les données et les traitements sont séparés, mais surtout, les données sont immuables. Chaque traitement, représenté par une fonction au sens informatique, produit un nouvel état en sortie plutôt que de modifier l'état en entrée. La validité de l'état en sortie est donc de la responsabilité unique du traitement courant.

Événements de domaine

Caractéristiques des événements du domaine

- Objet immuable : en tant qu'enregistrement fiable de quelque chose qui s'est produit dans le passé.
- Nommage : expression verbale au passé, issue du langage commun, telle que *EntretienPlanifié*.
- Contient un champ de type *timestamp* indiquant le moment précis de l'événement.
- Contient l'identifiant du/des *Entités/Agrégats* impliqués et les éventuelles données modifiées dans leur état.

> *« Something happened that domain experts care about. »*
>
> Eric Evans

Problématique 1 : expliquer un état

Une *Entité* est uniquement responsable de maintenir un état cohérent et intègre, tout au long de son cycle de vie. Lorsque l'on a besoin de savoir comment elle en est arrivée à son état actuel, c'est très difficile à reconstituer. C'est comme un enquêteur de police cherchant à élucider une scène de crime. Les pistes d'audit ou les historiques de changement d'état peuvent aider mais ont une visée plus généraliste.

Problématique 2 : état distribué

Cela se complexifie avec des systèmes distribués, en vogue avec les micro-services, dans lesquels on préfère jouer sur la cohérence pour garantir les autres paramètres : disponibilité et tolérance au partitionnement[51]. Du fait du temps de propagation de l'information, il peut donc se produire une désynchronisation temporaire de l'état entre les machines sur le réseau, alors même que sur chaque machine, l'état de l'objet est cohérent, garanti par les *Entités/Agrégats*. Les difficultés surviennent lorsqu'il faut réconcilier plusieurs changements sur un état, arrivés dans le désordre ou de sources de données distinctes (*Eventual Consistency* ou *Cohérence à terme* en français).

Modélisation par événements

Pour résoudre ces problématiques, la modélisation par état ne suffit pas. Il faut la compléter ou la remplacer par une modélisation par événements. L'activité dans le modèle se concrétise alors par une série d'événements discrets. Les événements sont alors des objets métier faisant pleinement partie du modèle de domaine. Ils sont la représentation de quelque chose qui s'est passé dans le domaine et qui parle aux experts métier. Le cœur d'un atelier d'*Event Storming* consiste à trouver ces événements et à les lier entre eux chronologiquement pour avoir une image globale d'un processus métier.

---

51. Cf. Théorème CAP : https://en.wikipedia.org/wiki/CAP_theorem

Attention à ne pas confondre les événements de domaine avec les événements système, plus techniques que fonctionnels, même s'ils peuvent se déclencher en même temps.

Découplage

Il est également possible d'utiliser les événements de domaine pour faire communiquer deux composants entre eux sans les coupler. Mais la technique ne doit pas être ce qui motive ce choix de design, d'autant que cela peut induire en erreur, car plus difficile à comprendre qu'un branchement direct et explicite. C'est le métier qui doit motiver l'introduction d'un tel événement dans le modèle. Le cas typique est un cas d'utilisation formulé en «Quand une commande est validée, un mail de confirmation est envoyé.»

Event Handler

Dans la pratique, il intervient un *Event Handler*, objet créé et actionné lors d'un événement qui lui est spécifique.

Plusieurs options d'implémentation sont possibles[52] :

- Les *Handlers* sont-ils déclenchés immédiatement ?
  - Si oui, le sont-ils au sein même de l'*Agrégat* ou juste au-dessus, au niveau des frontières de la transaction ? Dans les deux cas, le but est que l'événement soit traité immédiatement, de manière à avoir une opération atomique[53]. Ce type d'événements permet de faire communiquer plusieurs agrégats participant à la même opération, et ceci de manière découplée.
  - Sinon, on est en dehors d'une transaction, par exemple après un *commit*. Il s'agit alors de notification pour déclencher d'autres actions de manière asynchrone, telles que l'envoi d'un email, la propagation de l'information dans un système distribué, ou le déclenchement d'une opération dans d'autres *Bounded Contexts*. Dans ce dernier cas, on parle d'événements publics[54].

Autant les événements font partie intégrante du *Domaine*, autant leurs *Handlers* concernent la couche *Application* (cf. § *Architecture en couches* page 62). Ces derniers ne contiennent pas de logique métier, ils ne font que déléguer aux objets du domaine qu'ils coordonnent.

Persistance des événements

Il est possible de persister les événements et l'état en même temps. Cela peut être intéressant dans certains cas métiers. L'alternative est de ne stocker que les événements et de reconstruire l'état. Il faut alors que les événements le permettent. Pour plus d'informations, renseignez-vous sur l'*Event Sourcing*[55].

52. http://www.kamilgrzybek.com/design/how-to-publish-and-handle-domain-events/
53. L'opération ne réussit que si toutes ses composantes ont réussi
54. https://verraes.net/2019/05/patterns-for-decoupling-distsys-explicit-public-events/
55. https://martinfowler.com/eaaDev/EventSourcing.html

Architecture en couches

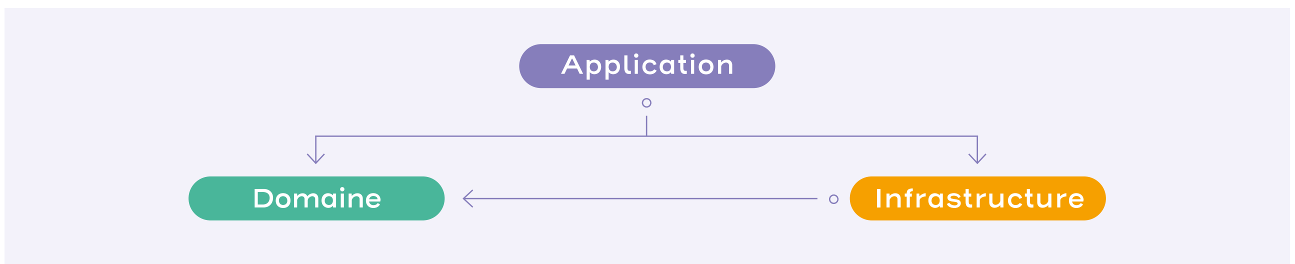
Lorsque les problématiques sont mélangées...

Lorsque l'on écrit une application, il peut arriver de mélanger les problématiques les unes avec les autres : de la logique métier se retrouve dans l'interface graphique et en base de données ; des logiques d'affichage et de persistance sont présentes dans des composants métier. Cela arrive car c'est la manière la plus simple de faire, du moins au début.

Ce mélange des problématiques reproduit dans une large base de code la rend difficile à comprendre et aussi fragile qu'un château de cartes, et de manière exponentielle à sa taille. La maintenance devient impraticable : si chaque endroit dans le code traite de plusieurs problématiques à la fois, il est facile de se mélanger les pinceaux ou de perdre le fil, pour finir par, au mieux, comprendre de travers ce que fait le code. La mise en place de bonnes pratiques de développement à mêmes d'améliorer la maintenabilité, est un vrai casse-tête.

Séparation en couches

Pour établir une claire séparation des responsabilités (*separation of concerns*), un programme un tant soit peu complexe doit être partitionné en couches. L'architecture DDD traditionnelle en mentionne trois : *Application*, *Domaine* et *Infrastructure*.



- **Le *Domaine* concentre tout ce qui est métier.** Il est isolé, et ne présente aucune dépendance. On le constate sur le schéma ci-dessus par le fait qu'il ne pointe vers aucune autre couche. Il est exempt de logique non-métier (logique applicative, logique d'interface, problématiques d'infrastructure).
- **La couche *Application* sert d'interface avec le monde extérieur,** directement ou au travers une couche de *Présentation*. La couche *Application* dépend des deux autres couches *Domaine* et *Infrastructure*, et en assure la coordination ainsi que l'isolation. Elle sert de masque, c'est-à-dire d'abstraction de ces couches internes vis-à-vis de l'extérieur. Du fait de ce dernier point, la couche *Application* est la traduction technique du *Bounded Context contenant tous les scénarios de cas d'utilisations*.
- **La couche *Infrastructure* est celle des rouages techniques.** Elle est en relation avec les éléments de plus bas niveau au cœur du système (système de fichiers, base de données, ORM, serveur SMTP...).

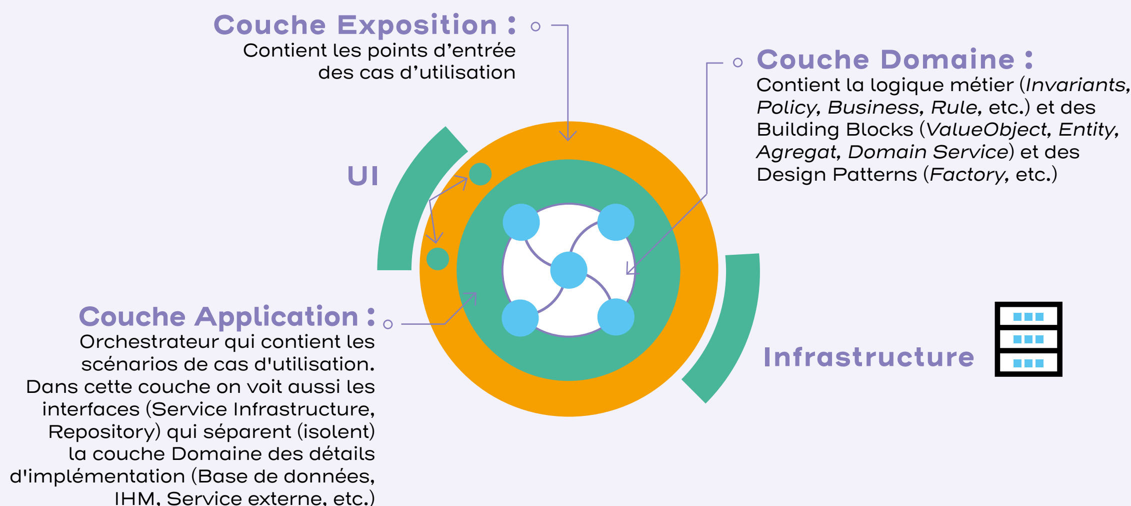
Des patterns architecturaux tels que *Ports and Adapters*[56], peuvent venir compléter cette architecture pour favoriser le découplage entre couches.

Avec une telle séparation, les objets métier ne sont pas responsables de leur affichage, ni de leur persistance. Ils ne sont pas pollués par une sémantique extérieure comme par exemple un framework. Ils peuvent se concentrer sur l'expression du modèle de domaine, déjà suffisamment complexe en soi.

Type de *Services* par couche

Les *Services* se retrouvent dans les trois couches de l'architecture :

- **Les *Services de Domaine* participent à la prise de décisions** pour réaliser l'opération métier. Les autres types de services ont accès au domaine mais ne sont pas acteurs dans la prise de décisions métier.
- **Les *Application Services* jouent le rôle d'intermédiaires et de traducteurs.** Ils masquent la complexité des couches sous-jacentes pour ne présenter qu'un contrat succinct et cohérent aux clients externes. Depuis l'extérieur, on ne peut pas manipuler directement les objets du domaine. On passe par des représentations sous la forme de *DTO*.
- **Les *Services d'Infrastructure* proposent une abstraction** des éléments de bas niveau adaptée à l'application.



56. https://alistair.cockburn.us/hexagonal-architecture/

04 Conclusion

Conclusion

Le DDD est une approche de développement logiciel qui remet les besoins métiers au cœur des préoccupations logicielles et répond particulièrement bien aux différentes problématiques rencontrées sur des projets et des domaines complexes ou stratégiques pour l'entreprise.

L'approche DDD permet d'éviter les applications « CRUD », généralement issues de développements rapides. Le CRUD est inadapté aux domaines complexes pour deux raisons principales : l'absence d'intention métier dans le code le rend délicat à maintenir et l'interface graphique réalisée est souvent perçue par les utilisateurs comme ni adaptée, ni pratique.

Le DDD n'est pas une technologie, un framework ou une méthode. Cette approche est indépendante des « buzz » dont est friande l'industrie du logiciel. Le DDD est un état d'esprit qui instille ses bénéfices des phases de cadrage aux étapes d'implémentation et qui fédère une diversité d'acteurs : décideurs, experts métiers, maîtrise d'ouvrage, responsables de domaine, équipes de développement, d'architecture…

La clé de sa réussite repose sur la collaboration entre tous ces acteurs, facilitée par un langage commun (Ubiquituous Language), une vision générale des contextes (Context Mapping), un ensemble de méthodes (ateliers Event Storming, Example Mapping…), de pratiques Craft (Behaviour-Driven Development, Mob Programming…) et de patterns (Agrégats, Value Objects…).

L'approche DDD est particulièrement intéressante grâce à sa facilité à intégrer les pratiques externes jusqu'aux plus récentes. Ainsi, les solutions d'architecture avec une ampleur métier comme l'Event Sourcing, les Microservices et les architectures réactives se conjuguent facilement avec le DDD grâce à sa focalisation sur la compréhension des problématiques métier.

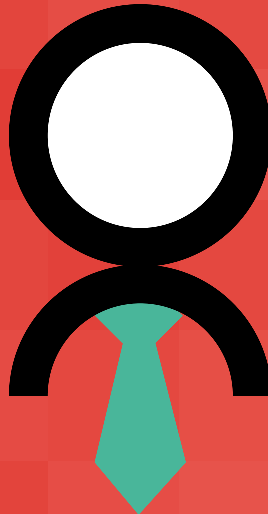
C'est en cela que le DDD n'a pas pris une ride depuis sa formalisation en 2003 dans le « Blue Book » d'Eric Evans. La communauté DDD est en continuelle évolution. De plus en plus d'entreprises utilisent cette approche et la considèrent comme un investissement rentable à moyen et long terme.

Ce n'est pas encore votre cas ? Ne soyez pas effrayés ou au contraire impatients de la mise en pratique de tous ces éléments. Aucun n'est strictement nécessaire ou spécifique au DDD. Comme une boîte à outils, prenez ce qui vous semble utile, pertinent et applicable dans votre organisation, le principal étant de bien garder en tête les principes clés du DDD. L'objectif est d'identifier où et quand le DDD peut vous apporter une nette plus-value car sa mise en place nécessite un investissement important pour monter en compétences, faire évoluer l'organisation et la culture dans l'entreprise.

On vous l'avoue : même si le DDD est relativement connu, on aimerait que cette approche soit plus largement mise en pratique dans notre industrie. Sa richesse et sa profondeur séduisent ceux qui ont franchi le pas. Un accompagnement peut faciliter sa prise en main et son adoption.

La communauté Software Craftsmanship et architecture de SOAT serait heureuse de vous accompagner autour de ces sujets, au travers de formations et de coaching, et vous invite à regarder ses autres publications correspondant à des sujets qui nous tiennent à cœur.

05 Retours d'expérience



Arthur Le Dref : Développeur chez Ceetiz

J'ai appris et expérimenté le DDD aux côtés de Sepehr pendant plus de 6 mois. Il m'en a montré les tenants et aboutissants tout en essayant de m'inculquer l'état d'esprit DDD. Car cela va bien au-delà d'un design d'architecture, ou d'un design pattern, le DDD nous emmène jusqu'à la gestion de projets, et comment mieux interagir avec le reste de l'entreprise.

Au premier abord, le DDD m'a paru très complexe. On utilise des termes spécifiques comme *Aggregate Root, Bounded Context, Ubiquitous Language*, etc. Pour un novice, cela fait beaucoup à apprendre d'un coup, et peut faire peur.

Cependant, en allant plus loin que ces termes, et en allant sur le sens même de ses principes, on en revient en fait au fondamental d'un projet (informatique ou non) : casser les silos des organisations, parler le même langage que les personnes qui font le travail tous les jours, travailler dans des contextes isolés du reste du système.

Le Domain Driven Design nous fait revenir aux fondamentaux de notre travail, nous pousse à sortir de nos bureaux et à faire face à la réalité de nos entreprises.

Ce design va vous paraître étrange au début, voire contre-intuitif. Cela va aller totalement à l'encontre de ce que l'on vous a appris à l'école (pas besoin de créer de l'abstraction entre des métiers qui n'ont aucun rapport, le copier-coller n'est donc plus interdit !!!), ou même de ce que vous avez pu expérimenter en entreprise pendant de nombreuses années (pas de mise en commun sous prétexte que ça se ressemble, si les métiers sont différents alors le code doit l'être aussi !).

Si vous lui en laissez l'opportunité, le DDD va vous étonner, et vous pourrez vous dire ensuite “Mais en fait il n'y a rien de sorcier, cela montre juste la base, ce que nous avons en fait oublié !”.

J'ai appris, en suivant cette technique, à découper et découpler mon code, à utiliser des termes qui ont du sens au sein de mon entreprise, au point que mon code pourrait être montré à quelqu'un sans aucune connaissance technique, et qu'il pourrait comprendre ce qu'il se passe. J'ai aussi compris que si deux notions métiers n'avaient rien à voir, alors leur code devrait être à deux endroits différents et suivre chacun leur vie sans impacter celle de l'autre. C'est ce que l'on appelle un *Bounded Context* en DDD. Cela vous rappelle-t-il quelque chose ? Et oui, cela fait écho au S de *SOLID*[57], le Single Responsability Principle, qui prône la séparation du code en fonction de la responsabilité d'une classe, qui n'est autre que son contexte métier.

Le DDD a été d'un grand apport pour moi, car en partant d'une *codebase Legacy* très complexe, l'équipe dont je fais partie a pu faire sortir l'essence de ce qui avait été entrepris, avec un code plus clair et décorrélé de l'existant.

---

57. https://fr.wikipedia.org/wiki/SOLID_(informatique)

Cela a été compliqué, je ne le cache pas. Comme nous étions novices du DDD nous n'avions pas forcément compris tout les intérêts, et nous avons aussi fait des erreurs.
Nous étions en fait sortis de la logique métier, et étions restés la tête dans la technique. Et c'est une chose à laquelle nous devons tous faire attention, et que le DDD essaie d'éviter.

Parler technique est facile (pour des développeurs). Comprendre ce qui est fait tous les jours par les autres employés de nos entreprises est un frein.

Le DDD veut réunir les développeurs et les autres personnes de l'entreprise. Une réunion d'évolution de système ne devrait jamais se faire sans les développeurs. En effet, en passant par un intermédiaire, des données sont toujours perdues, des termes sont changés et à la fin, c'est le système qui dérive vers quelque chose d'inattendu pour les utilisateurs, et du temps est perdu alors que le développement du projet n'a même pas commencé.

Il faut se rappeler que l'objectif de notre travail est de simplifier celui des autres, pas de le tordre dans notre vision partielle des choses.

Mais outre les notions de gestion dans une entreprise ou dans un projet, le DDD apporte un certain nombre d'outils pour nous faciliter la vie lors du développement d'un applicatif, du design pattern au design d'architecture.
Tous ces outils peuvent être d'abord laissés de côté afin d'explorer tous les tenants et aboutissant du DDD. Cela permet de simplifier et de redonner du sens au code produit. Une fois cela fait, on se rend compte des limites de nos connaissances, et les outils apportés par le DDD prennent alors le relai pour nous aider à aller plus loin.

Vous pourrez d'ailleurs trouver énormément de similitudes entre l'architecture hexagonale et une architecture DDD, car les principes sont finalement les mêmes, bien qu'explicités en des termes différents.

En lisant le code on peut comprendre directement ce qu'il se passe car il n'y a pas de magie d'un framework, et on parle tout de suite comme les sachants métiers car ce sont leurs termes qui sont dans le code. Et cela a beaucoup d'avantages, notamment pour les nouveaux arrivants sur le projet. On a besoin de moins de temps d'adaptation, et on est opérationnel plus rapidement.

J'ai pu le voir sur un projet fait en DDD. Un nouveau est arrivé dans l'équipe, et qui plus est un junior. Je lui ai demandé de faire une modification sur le projet, sans l'avoir informé de rien. Je lui ai juste dit de regarder le code et d'essayer de se débrouiller tout seul. Il a pu ainsi s'imprégner du métier, mais aussi directement faire les bonnes modifications. Il a pu aussi par la suite faire une réunion de présentation avec les sachants métiers et était à l'aise avec les termes utilisés par eux. Après avoir débriefé avec lui sur ce qu'il avait pensé du travail qu'il venait d'accomplir ainsi que de la réunion, il était assez choqué d'avoir pu parler directement sans être perdu. Il m'a dit qu'il avait trouvé le format de notre architecture bizarre, et que cela l'avait perturbé, mais que maintenant qu'il avait totalement complété sa tâche il comprenait en fait pourquoi nous avions fait cela, et qu'il allait se mettre au DDD plus sérieusement car il en comprenait les bénéfices.

Sepehr NAMDAR : Développeur chez Canal+

L'équipe Data de Canal+ utilise un progiciel qui lui permet de remonter des offres adaptées à un client en fonction de ses appétences grâce à un arbre de décision.

Les services exposés par ce progiciel sont complexes à comprendre car ils sont constitués d'un langage technique propre à lui-même et difficilement compréhensible par les systèmes en aval. L'inconvénient est que chaque client doit étudier et comprendre le mode de fonctionnement de ces services pour pouvoir communiquer avec eux.

Au moment de mon arrivée dans cette équipe, il n'y avait qu'un seul client pour ce progiciel et ma tâche était de faire en sorte que d'autres types de clients puissent utiliser les services exposés par le progiciel.

Nous avons donc commencé par mettre en place une API de type Open Host Service (OHS) afin de satisfaire les demandes de tous les clients, en leur cachant la complexité qui se trouve derrière ce progiciel. De plus, il nous a fallu adapter chaque service au langage métier de chaque client, ce qui a permis aux clients existants de se débarrasser de la couche d'anti-corruption (ACL) qu'ils avaient mise en place et de rendre facile la communication avec le progiciel.

Autre problème que j'ai remarqué en arrivant dans cette équipe, c'était la communication entre les différents acteurs du projet. En effet, par exemple lors d'un *Daily Scrum Meeting*, nous avions souvent du mal à comprendre les tâches en cours de chacun.

J'ai donc commencé par organiser des sessions d'*Event Storming* afin d'aligner la compréhension métier de chacun des membres du projet au même niveau et de construire un *Ubiquitous Language* au sein de notre équipe. Nous avons ensuite utilisé ces termes métier dans notre code, à tel point que notre *Product Owner*, qui n'était pas du tout technique, était capable de lire le code et de le comprendre.

Thibaut CANTET : Développeur à la Fédération Française de Tennis

Je m'intéressais au DDD depuis un moment sans l'avoir mis en place, lorsque je suis arrivé à la FFT pour travailler sur le site de la billetterie de Roland-Garros.

Le code legacy était une base de code propre d'un point de vue couverture de tests mais avait une architecture en couches fortement couplées de bas en haut.

Le changement de prestataire de paiement ou PSP (*Payment System Provider*) a été l'occasion de refondre complètement un grand pan de l'application web.

Au lieu de reprendre le code existant et de le modifier, j'ai préféré repartir de zéro en isolant complètement le nouveau code (dans un package différent à la racine de l'application).

La partie métier du code présentait de nombreux mono-modèles utilisés depuis la couche d'accès aux données jusqu'à l'API REST, et partagés dans plusieurs contextes différents (tunnel d'achat, espace "mes commandes"...). Cela engendrait des classes très importantes mélangeant différentes notions. Les règles métiers étaient disséminées un peu partout dans les services, les entités...

Pour intégrer le DDD, j'ai d'abord commencé à mettre en place des *Bounded Contexts*. Pour cela, j'ai préféré d'abord dupliquer des classes par contexte, matérialisées par des packages différents. Pour identifier ces contextes, je me posais simplement la question "Est-ce que l'on parle de la même chose ? Est-ce que l'on est dans une même transaction métier ? Est-ce que je suis dans le même cycle de vie ?"

Modèles, repositories, services ont été copiés/collés dans plusieurs endroits du code. Après avoir connu *DRY (Don't Repeat Yourself)* et factorisé à outrance, cela fait un peu bizarre !

Mais petit à petit, je me suis rendu compte que les modèles qui semblaient être la même chose n'évoluaient pas dans la même direction, et pas au même rythme. Les besoins n'étaient pas les mêmes. Je n'avais plus peur de faire des changements, car ils étaient alors isolés dans du code spécialisé.

Plus ma connaissance métier s'approfondissait, plus je renommais mes entités, repositories et interfaces avec des termes utilisés par l'équipe métier. L'*Ubiquitous Language* fut donc la seconde étape de la mise en place de DDD.

L'organisation d'un *Event Storming* m'a permis de vérifier la compréhension des problématiques métier et de préciser certains termes de vocabulaire.

De même, la rédaction des tests devenait plus précise en se focalisant beaucoup plus sur la complexité métier que sur les problèmes techniques. Les *Value Objects* ont été l'étape suivante. Il existait notamment des identifiants suivant certaines règles de sérialisation. Les *Value Objects* ont permis de centraliser la logique de sérialisation/désérialisation et leur validité (*invariant*).

Le dernier paradigme que j'ai implémenté est la mise en place d'événements de domaine permettant de propager dans mon application le résultat de différentes opérations exécutées. Cela a été facilité par la mise en place d'une architecture *CQRS*[58] *(Command and Query Seggregation Responsability).*

J'ai toujours été convaincu qu'il est nécessaire de partir du besoin avant de démarrer les développements. J'ai donc facilité la proximité entre mon équipe de développement et le métier afin de répondre le plus précisément à leur exigences. Cela s'illustre très souvent en posant des questions de vive voix au métier, plutôt que de passer par des outils qui déforment la compréhension et rendent la communication moins efficace, ou pire encore, en supposant une réponse sans en poser la question. DDD m'a finalement beaucoup aidé à faire en sorte que mon code soit robuste, fasse le minimum, mais parfaitement.

La *Clean Architecture*[59] ou l'*Hexagonal Architecture* apportent des lignes directrices pour organiser son application et isoler son domaine, mais ne donnent pas beaucoup d'éléments pour organiser son code métier, alors que DDD aide dans le quotidien afin que le code soit le plus explicite possible, compréhensible et reflète les enjeux métiers de chaque application.

58. https://www.soat.fr/publications/cqrs-event-sourcing
59. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Auteurs

Sepehr NAMDAR FARD

Développeur, domaine de compétences Java EE.

Ayant eu l'occasion de travailler sur des missions variées, que ce soit de la pure maintenance ou le suivi complet d'un projet, j'ai pu acquérir des connaissances sur toutes les phases d'un développement logiciel. La réussite d'un projet dépend en grande partie des méthodologies appliquées lors de son déroulement. Celles-ci sont souvent l'Agilité, le Lean, le Behaviour Driven Development et le Clean Code.

Romain DENEAU

Développeur full stack .NET TypeScript – 18 ans d'expérience

Un temps passé chef de projet, je suis désormais revenu à la passion qui m'anime depuis longtemps, le développement. Craftsman convaincu, je participe au partage des bonnes pratiques et du simple design aux moyens d'articles et des MasterClasses Craft chez SOAT.

Relecteurs

Arthur LE DREF

Développeur, j'aime intervenir sur tous les aspects d'un projet, de l'infrastructure jusqu'au Front.

Passionné de beau code, j'ai principalement travaillé sur des applications Legacy dans le monde Java. J'ai tiré de cela une grande expérience sur les mauvais choix de design qui me permet aujourd'hui de les identifier au plus tôt, et de savoir comment en revenir à moindre coût. DDD, Clean Code et Pragmatism-Driven-Design sont pour moi les clés de la réussite d'un projet.

Thibaut CANTET

Développeur, software crafter, formateur.

J'ai pu travailler sur de nombreux projets dans des contextes très variés, de la startup jusqu'aux groupes du CAC 40, sur des projets green fileld ou sur du code Legacy.

La compréhension des besoins fonctionnels, les tests et le design, a toujours été un élément central dans mon travail. DDD, les pratiques craft et l'agilité m'aident au quotidien pour développer des applications complexes dans des contextes incertains.

SOAT

→ Digitalize society

90, quai Panhard et Levassor_75013 Paris
T (33) 1.44.75.42.55
contact@soat.fr

SOAT.FR