MASTER EN SCIENCES INFORMATIQUES

Domain Driven Design comme facteur d'évolution des modèles mentaux partagés dans le développement logiciel

ZENOBI, Jérôme

Award date:
2023

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022-2023

Domain Driven Design comme facteur d'évolution des modèles mentaux partagés dans le développement logiciel

Jérôme ZENOBI

..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : Benoit VANDEROSE

Co-promoteur : Julie HENRY

Mémoire présenté en vue de l'obtention du grade de Master 60 en Sciences Informatiques

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Remerciements

Tout d'abord, je tiens à exprimer ma plus sincère gratitude à mon promoteur, Benoît Vanderose, et à ma co-promotrice, Julie Henry, pour leur guidance experte, leur soutien et leur encouragement tout au long de ce projet. Leur enthousiasme pour le sujet et leur dévouement à l'égard de leurs étudiants ont été une véritable source d'inspiration pour moi.

Je tiens à remercier chaleureusement tous les participants de mon étude. Leurs contributions ont été inestimables et ce travail n'aurait pas pu être réalisé sans leur aide.

Un grand merci à ma merveilleuse épouse, Aurore, pour son amour inébranlable et son soutien constant. Sa patience et son encouragement ont été mon pilier tout au long de ce voyage.

Je voudrais également exprimer ma gratitude envers ma famille. Leur amour inconditionnel et leur soutien inébranlable ont été essentiels tout au long de ce parcours.

Je tiens à exprimer mes remerciements au groupe des copains. Votre amitié et vos encouragements ont souvent été l'énergie dont j'avais besoin pour continuer à avancer.

Enfin, une pensée pour Brandon Sanderson, dont le récent achèvement de la saga "La Roue du Temps" a constitué une source d'inspiration amusante et bienvenue pendant les longues heures passées à travailler sur ce mémoire. Comme il a réussi à terminer sa saga épique, j'ai été encouragé à faire de même avec ce travail de fin d'études !

Résumé : Présents depuis toujours et dans tous les domaines, les modèles mentaux sont des systèmes qui nous permettent de mieux comprendre les concepts qui nous entourent. Ces modèles sont des outils nous permettant de résoudre plus facilement des problèmes efficacement et ainsi prendre de meilleures décisions dans notre quotidien comme dans notre vie professionnelle.

Ce mémoire examine les modèles mentaux partagés dans le contexte des développements logiciels en entreprise. Afin de mieux comprendre les problèmes que peuvent rencontrer les équipes de développement pendant les différentes phases d'un projet nous allons voir comment les modèles mentaux partagés influencent le développement d'un logiciel ainsi que l'impact du Domain Driven Design sur ces modèles.

Mots-clés *: shared mental models - programming projects - software development - agile - domain driven design*

Abstract :

Present since the beginning of time and in all fields, mental models are systems that allow us to better understand the concepts that surround us. These models are tools that allow us to solve problems more easily and efficiently and thus make better decisions in our daily lives as well as in our professional lives.

This thesis examines shared mental models in the context of software development in the enterprise. In order to better understand the problems that development teams may encounter during the different phases of a project, we will see how shared mental models influence the development of software and the impact of the Domain Driven Design approach on these models.

Keywords *: shared mental models - programming projects - software development - agile - domain driven design*

Glossaire

— **Aggregate** : En DDD, un Aggregate décrit un groupe de données qui doivent être traitées ensemble en tant qu'unité logique. Il est utilisé pour modéliser les relations entre les différents objets métiers d'une application.

— **Aggregate Roots** : En DDD, un Aggregate Root est l'Entity principale d'un Aggregate, utilisée pour gérer les opérations et les relations entre tous les objets associés de l'Aggregate. Il maintient l'intégrité de l'Aggregate et de ses objets associés, et est responsable de la validation des données et de la mise à jour des objets associés.

— **Artefact** : Les artefacts sont des produits tangibles ou des documents créés durant les différents moments du cycle de vie du projet Agile, à savoir la planification, l'exécution, la révision et la clôture. Les artefacts les plus couramment utilisés sont le Product Backlog, les User Stories, les Sprint Backlogs, les Burndown Charts et les Definition of Done.

— **Backlog** : Le Backlog est la liste ordonnée (priorisée) des besoins (généralement formulés sous forme de User story) du projet[1].

— **Bounded Context** : En DDD, un Bounded Context est une limite de compréhension qui définit les termes et les modèles de données utilisés dans un système donné. Il est utilisé pour découpler les différents modèles de données de l'entreprise et éviter les confusions entre les termes utilisés dans différents contextes.

— **Burndown Charts** : En méthode agile, les Burndown Charts sont des graphiques montrant la progression du travail pendant le Sprint en cours.

— **Codage** : Le codage dans le cadre d'une recherche qualitative se réfère au processus de l'organisation et de la catégorisation des données recueillies lors de l'étude. C'est une étape essentielle pour l'interprétation et l'analyse des données. Le processus de codage comprend généralement les étapes suivantes : codage ouvert, codage axial et codage sélectif.

— **Context Map** : En DDD, un Context Map est un outil qui permet de visualiser les relations entre les différents Bounded Contexts d'un système. Il sert à clarifier les relations entre les différents modèles de données et les termes utilisés dans les différents contextes.

— **Daily meeting** : La Daily meeting est une réunion quotidienne de moins de 15 minutes permettant à la Dev Team de se synchroniser, d'identifier les obstacles éventuels et de mesurer son avancement sur le Sprint en cours[1].

— **DDD** : Domain Driven Design

— **Definition of Done** : En méthode agile, il s'agit d'un ensemble de critères permettant de déterminer quand une tâche est considérée comme terminée.

— **Domain** :En DDD, un Domain est un ensemble de compétences et de connaissances spécifiques à une entreprise ou un secteur d'activité. Il décrit les problèmes spécifiques, les règles d'affaires, les terminologies et les processus qui caractérisent un domaine donné.

Le Domain est le cœur du DDD. Il est utilisé pour identifier les besoins métiers de l'entreprise et pour créer des modèles de données qui reflètent ces besoins. Les experts métiers et les développeurs travaillent ensemble pour comprendre le Domain et pour créer des logiciels qui répondent aux besoins réels de l'entreprise.

— **Domain-Specific Language** : Un DSL est un langage de programmation ou de modélisation conçu pour résoudre des problèmes spécifiques dans un domaine d'application particulier. Il est spécifiquement adapté pour exprimer les concepts, règles et processus de ce domaine de manière concise et facile à comprendre.

— **Entity** : Une Entity est un objet métier qui représente une chose unique et identifiable dans le domaine. Il est caractérisé par un identifiant unique qui le distingue des autres objets métiers. Les Entities peuvent avoir des propriétés et des comportements qui décrivent leur état et leur comportement dans le système.

— **Epic** : En méthode agile, une Epic est une fonctionnalité ou un ensemble de fonctionnalités majeures qui apportent une valeur significative à l'utilisateur final. Elle représente un objectif commercial ou une initiative importante pour l'organisation. Elles sont donc décomposées en de plus petites fonctionnalités appelées User Stories, qui sont ensuite développées et livrées en itérations.

— **Event Storming** : L'Event Storming est une méthode de modélisation collaborative qui permet de découvrir, de décrire et de comprendre les processus métier et les interactions entre les différents acteurs d'un système d'information. Il s'agit d'une technique itérative et interactive, qui implique la participation d'experts métiers et techniques.

— **Immutable** : En informatique, Immutable désigne un objet qui ne peut pas être modifié une fois créé. Il s'agit d'un état figé, qui ne peut pas être altéré après sa création. Les objets immutables sont souvent utilisés pour garantir la sécurité des données, en empêchant les modifications accidentelles ou malveillantes.

— **Mutable** : En informatique, Mutable désigne un objet qui peut être modifié après sa création. Il est opposé à l'objet immutable qui une fois créé ne peut pas être modifié. Les objets mutables sont souvent utilisés pour des besoins de flexibilité dans les traitements : ils permettent de changer leur état à tout moment.

— **Product Backlog** : En méthode agile, le Product Backlog est une liste des fonctionnalités ou des éléments de travail prioritaires pour le projet.

— **Product Owner** : Représentant des utilisateurs qui porte la vision du produit que l'on souhaite réaliser dans le cadre du projet. Il est responsable du Product Backlog et en interaction directe avec la Dev team [1].

— **Scrum Team** : La Scrum Team rassemble la Dev Team, le Product Owner et le Scrum Master.

— **Service** : En DDD, un service est un objet métier qui encapsule une logique spécifique à un domaine. Il est utilisé pour encapsuler les règles d'affaires complexes et les algorithmes qui ne peuvent pas être directement associés à une Entity ou un aggregate. Il est généralement utilisé pour effectuer des tâches spécifiques ou pour fournir une fonctionnalité spécifique à l'application.

— **Sprint Backlog** : En méthode agile, le Sprint Backlog est la liste des tâches à réaliser pendant le sprint en cours.

— **Systematic Mapping Study** : Le méthodologie SMS consiste en "un processus d'identification, de catégorisation et d'analyse des littératures existantes pour un certain sujet de recherche" [2]. L'objectif de cette méthode est d'obtenir une vue globale sur un sujet particulier, d'en présenter une évaluation impartiale et complète, de déterminer les possibles lacunes présentes dans les cette littérature et de donner d'éventuels futures pistes de recherche [3].

— **Tactical Pattern** : En DDD, les Tactical Patterns (ou modèles tactiques) sont des solutions éprouvées pour résoudre des problèmes de conception spécifiques rencontrés dans la mise en œuvre du DDD.

— **Ubiquitous Language** : L'Ubiquitous Language est un concept-clé du DDD. Il s'agit d'un langage commun et partagé par tous les membres de l'équipe de développement ainsi que les experts métiers impliqués dans le projet.

— **User Story** : Technique permettant de formaliser synthétiquement les besoins sans perdre de vue l'essentiel : le besoin concerne QUI, en QUOI il consiste et dans quel BUT [1].

— **Value Object** : En DDD, un Value Object (ou objet de valeur) est un concept-clé de modélisation, à savoir un objet qui n'a pas d'identité propre, mais qui est défini par sa valeur.

  Contrairement aux Entities qui sont identifiées par une clé unique, les Value Objects ne sont pas identifiés par une clé, mais plutôt par leur contenu. Une date de naissance ou un montant d'argent sont des exemples d'objets de valeur.

— **Vélocité** : A la fin d'une itération, l'équipe additionne les estimations associées aux User Stories qui ont été terminées au cours de cette itération. Ce total est appelé vélocité[4].

Table des matières

1 Introduction

L'utilisation des modèles mentaux partagés dans le développement logiciel est de plus en plus répandue dans les entreprises. Consciemment ou non, les développeurs travaillant quotidiennement sur un projet commun à long terme tendent naturellement à se rejoindre sur des concepts communs afin d'accélérer et d'améliorer la qualité de leur travail. Néanmoins, nous remarquons que certains outils et méthodologies mis en place sur les projets peuvent diminuer le temps nécessaire aux développeurs pour atteindre ces modèles mentaux partagés, et aider ainsi à atteindre plus rapidement les objectifs fixés par l'équipe.

Le début des années 2000 marque l'avènement d'un certain nombre de méthodologies dont l'objectif est d'améliorer les cycles de production et les procédés de collaboration. Figures de proue de ces nouveaux concepts, les méthodes Agile — telles que Scrum[5] ou eXtreme programming[6] — s'essayent à révolutionner l'approche des projets de développement logiciel.

Malgré la popularité des méthodes Agile, certains inconvénients subsistent et peuvent impacter les développements logiciels en entreprise. Parmi ces défauts figurent le manque de documentation, les difficultés de planification à long terme et l'adaptation aux projets de grande envergure. En outre, les méthodes agiles peuvent parfois négliger certains aspects techniques ou de qualité, et engendrer une charge de travail inégale pour les membres de l'équipe. Il est donc important d'être conscient des limites des méthodes agiles et d'adapter les pratiques en fonction.

Ainsi, de nouvelles approches sont apparues pour compléter et renforcer les pratiques Agile. En 2003, Mark Evans évoque pour la première fois le Domain-Driven Design (DDD), mettant l'accent sur le domaine métier et la logique associée. Cette approche vise à améliorer la compréhension et la modélisation des problématiques métiers, afin de faciliter le développement et la collaboration au sein des équipes Agile.

En dépit de sa méconnaissance par le grand public, le potentiel du DDD pour améliorer la collaboration et la compréhension des problématiques métiers dans le développement logiciel suscite un intérêt croissant dans la communauté de développement. Notre étude vise à explorer le lien potentiel entre le DDD et l'émergence de modèles mentaux partagés au sein des équipes de développement, ainsi qu'à identifier les mécanismes par lesquels le DDD facilite cette émergence.

La recherche se focalise sur deux points clés : d'une part, l'impact du DDD sur les modèles mentaux partagés, sans considération de la méthodologie utilisée ; et d'autre part, l'effet combiné du DDD avec les méthodologies Agile sur l'évolution de ces mêmes modèles.

L'objectif est de comprendre comment l'adoption du DDD et sa synergie avec les méthodologies Agile peuvent encourager des modèles mentaux partagés solides et cohérents au sein des équipes de développement logiciel, améliorant ainsi leur performance et leur collaboration.

Pour répondre à cette problématique, nous posons deux questions de recherche :

— La première question examine comment l'application du DDD dans le développement logiciel influence l'évolution des modèles mentaux partagés entre les différents acteurs impliqués.

— La seconde question s'intéresse à la manière dont l'application conjointe du DDD et des méthodologies Agile influence l'évolution des modèles mentaux partagés parmi les membres des équipes de développement.

Notre travail sera organisé de la manière suivante :

— Le second chapitre donnera les bases de l'étude des modèles mentaux partagés, présentera les méthodes agiles et le DDD.

— Le troisième chapitre listera une sélection d'ouvrages et d'articles portant sur les modèles mentaux partagés en développement logiciel, sur les projets qui adoptent les méthodes agiles, ainsi que sur les études traitant du DDD.

— Le quatrième chapitre définira notre problématique et nos questions de recherche, en mettant l'accent sur l'interaction entre le DDD et les modèles mentaux partagés.

— Le cinquième chapitre détaillera la méthodologie que nous avons utilisée pour récolter les données nécessaires à notre étude, ainsi que les techniques d'observation, d'entretien et de collecte de données.

— Le sixième chapitre décrira les résultats obtenus suite à la récolte des données, en analysant les tendances et les corrélations observées entre le DDD et les modèles mentaux partagés.

— Le septième chapitre répondra aux questions de recherche en discutant des implications de nos résultats pour les équipes de développement et les organisations qui adoptent le DDD.

— Enfin, nous conclurons notre travail en dressant un bilan sur les impacts des modèles mentaux partagés sur les équipes de développement, qu'ils soient bénéfiques ou préjudiciables. Nous soulignerons également certaines pistes de recherche futures permettant d'approfondir la compréhension des relations entre le DDD et les modèles mentaux partagés.

2 Contexte théorique et conceptuel

Ce chapitre vise à introduire les trois concepts au cœur de notre sujet d'étude : les modèles mentaux partagés (Cfr. 2.1), les méthodes agiles (Cfr. 2.2) et le DDD (Cfr. 2.3). Il ne s'agit pas d'être exhaustif, mais bien de donner quelques éléments indispensables à la bonne compréhension de notre travail.

2.1 Modèles mentaux partagés

La psychologie cognitive étudie la manière dont les individus comprennent leur environnement grâce à des "structures de connaissances" appelées modèles mentaux. Ceux-ci constituent des représentations internes de l'environnement qui servent à décrire, expliquer et anticiper des situations futures. Klimoski et Mohammed (1994)[7] ont élargi ce concept au niveau des groupes en introduisant l'idée de modèles mentaux partagés.

"Team mental models are team members' shared, organized understanding and mental representation of knowledge about key elements of the team's relevant environment. They are a mechanism through which team members can develop shared understanding of task requirements, team interaction patterns, and task strategies"

– Klimoski et Mohammed[7]

Les modèles mentaux partagés correspondent aux représentations mentales communes que des individus ont à propos d'un sujet ou d'une situation particulière. Ces modèles sont façonnés par les expériences, les connaissances et les croyances de chaque personne et peuvent être conscients ou inconscients[8].

Le partage de modèles mentaux implique que les individus possèdent des représentations mentales similaires quant à la manière dont une situation devrait se dérouler ou évoluer. Ces modèles peuvent se construire grâce à des interactions sociales, des discussions, des expériences communes ou d'autres formes de communication.

Les modèles mentaux partagés revêtent une importance considérable, car ils permettent aux individus de collaborer de façon plus efficace et de prendre des décisions mieux informées. Lorsque les membres d'une équipe partagent des modèles mentaux, ils sont capables de communiquer plus aisément, de comprendre les points de vue des autres et de coopérer pour atteindre des objectifs communs.

Toutefois, il convient de reconnaître les limites des modèles mentaux partagés. En effet, il serait erroné de supposer qu'en apprenant un nouveau modèle, celui-ci devient universel et applicable à tous les domaines[9]. Il est donc essentiel de considérer les spécificités de chaque domaine et d'adapter les modèles mentaux en conséquence.

Après avoir introduit le concept des modèles mentaux partagés et leur importance dans la collaboration et la prise de décision, il est essentiel d'examiner les facteurs qui influencent leur développement au sein d'une équipe(Cfr. 2.1.1). Cette section traitera des éléments-clé qui contribuent à la construction et à l'évolution des modèles mentaux partagés : la communication, la formation, les expériences communes et le contexte organisationnel. En analysant ces facteurs, nous comprendrons mieux comment les équipes parviennent à développer des représentations mentales communes et comment ces représentations peuvent être renforcées ou modifiées au fil du temps.

Ensuite, il est crucial d'examiner les méthodes de mesure des modèles mentaux partagés (Cfr. 2.1.2. L'évaluation de ces modèles fournit une indication de leur portée et de leur applicabilité dans un contexte spécifique. Cette section discutera des différentes méthodes d'évaluation des modèles mentaux partagés, comme les questionnaires, les entretiens, les observations et les simulations, ainsi que leurs avantages et leurs inconvénients. La compréhension des méthodes de mesure est cruciale pour étudier l'impact des modèles mentaux partagés sur la performance de l'équipe et pour mettre en œuvre des interventions visant à améliorer leur développement.

Enfin, nous aborderons l'impact des modèles mentaux partagés sur la performance de l'équipe (Cfr. 2.1.3). Cette section explorera les relations entre les modèles mentaux partagés, la coordination, la communication et la prise de décision au sein des équipes, et comment ces facteurs contribuent à améliorer la performance globale. En étudiant l'effet des modèles mentaux partagés sur la performance des équipes, nous pourrons mieux comprendre comment optimiser la collaboration et la prise de décision dans divers contextes de travail, y compris le développement logiciel.

2.1.1 Facteurs influençant

Le développement des modèles mentaux partagés au sein d'une équipe est influencé par plusieurs facteurs qui interagissent de différentes manières. Le leadership est un élément essentiel, car un leader efficace facilite la communication et la collaboration, encourageant ainsi le partage d'informations et la formation de modèles mentaux partagés[10]. Un leader qui soutient l'apprentissage collectif et favorise un environnement de confiance et d'ouverture permet aux membres de l'équipe de partager leurs perspectives et d'aligner leurs modèles mentaux.

La formation conjointe des membres de l'équipe, en particulier lorsqu'elle est axée sur les objectifs communs et les tâches interdépendantes, peut également renforcer les modèles mentaux partagés[11]. La formation doit être adaptée aux besoins spécifiques de l'équipe et doit inclure des occasions de partager et de discuter des expériences et des connaissances. De plus, la composition et la structure de l'équipe peuvent influencer la façon dont les modèles mentaux se forment et se partagent. Des équipes de taille réduite et une répartition claire des rôles et responsabilités favorisent une meilleure compréhension des attentes et des objectifs communs[12].

La diversité des membres de l'équipe, en termes de compétences, d'expériences, de cultures et de personnalités, peut enrichir les modèles mentaux partagés en apportant des perspectives variées[13]. Cependant, une diversité excessive peut parfois compliquer les processus de communication et augmenter les risques de malentendus. Il est donc crucial de trouver un équilibre et de gérer activement la diversité pour faciliter le développement de modèles mentaux partagés.

Les mécanismes de communication au sein de l'équipe jouent également un rôle essentiel dans le développement des modèles mentaux partagés[14]. Une communication ouverte, régulière et bidirectionnelle favorise la circulation des informations et la compréhension mutuelle. Les outils et les technologies de communication appropriés peuvent également faciliter l'échange d'informations et la collaboration à distance.

L'expérience de travail en équipe influence également le développement des modèles mentaux partagés. Les membres de l'équipe ayant une expérience préalable de travail en équipe sont plus enclins à développer des modèles mentaux partagés, car ils comprennent mieux les dynamiques de groupe et les attentes mutuelles[15]. Les équipes qui travaillent ensemble depuis longtemps ont également tendance à développer des modèles mentaux partagés plus solides.

Enfin, l'alignement des objectifs et des valeurs est un facteur-clé pour le développement des modèles mentaux partagés. Lorsque les membres de l'équipe partagent des objectifs et des

valeurs communs, ils sont plus susceptibles de développer des modèles mentaux partagés[16]. L'alignement des objectifs facilite la collaboration et la coordination, tandis que des valeurs partagées contribuent à renforcer la confiance et l'engagement au sein de l'équipe.

2.1.2 Mesures

La mesure des modèles mentaux partagés est essentielle pour comprendre et améliorer la performance des équipes. Plusieurs méthodes ont été développées pour évaluer les modèles mentaux partagés, chacune ayant ses avantages et ses inconvénients.

Les questionnaires auto-rapportés sont l'une des méthodes les plus couramment utilisées pour mesurer les modèles mentaux partagés[17]. Les participants sont invités à répondre à des questions sur leur compréhension des tâches, des objectifs, des rôles et des responsabilités au sein de l'équipe. Ces questionnaires peuvent fournir des informations précieuses sur les perceptions individuelles et collectives, mais ils sont également sujets aux biais de réponse et à l'autocensure.

Les entretiens et les discussions de groupe sont une autre méthode utilisée pour évaluer les modèles mentaux partagés[18]. Cette approche permet une exploration en profondeur des représentations mentales des membres de l'équipe et de leurs interactions. Toutefois, ces méthodes peuvent être coûteuses en temps et en ressources, et les résultats peuvent être influencés par des facteurs tels que les dynamiques de groupe et la présence de l'intervieweur.

Les techniques de cartographie conceptuelle, telles que la cartographie cognitive ou les réseaux sémantiques, sont également utilisées pour mesurer les modèles mentaux partagés[19]. Ces techniques impliquent la représentation graphique des concepts et des relations entre eux, ce qui permet une analyse visuelle des structures de connaissances partagées. Bien que ces méthodes fournissent une représentation visuelle claire des modèles mentaux, elles peuvent être difficiles à interpréter et à quantifier.

Les simulations et les jeux de rôle peuvent également être utilisés pour évaluer les modèles mentaux partagés en contexte[20]. Les participants sont placés dans des situations réalistes et leurs interactions et leurs performances sont observées pour déterminer leur degré de partage des modèles mentaux. Ces méthodes permettent une évaluation plus écologique des modèles mentaux partagés, mais elles peuvent être complexes à mettre en œuvre et à analyser.

Enfin, des méthodes plus récentes, telles que l'analyse des réseaux sociaux[21], ont été appliquées pour étudier les modèles mentaux partagés en examinant les relations et les interactions entre les membres de l'équipe. Ce type d'approches offre une perspective unique sur les modèles mentaux partagés, mais nécessite des compétences et des outils spécifiques pour l'analyse des données.

2.1.3 Performance d'équipe

L'impact des modèles mentaux partagés sur la performance de l'équipe a été étudié dans de nombreux contextes et domaines. Les recherches montrent que les modèles mentaux partagés peuvent avoir un effet significatif sur la performance de l'équipe en améliorant la communication, la coordination et la prise de décision[22, 14, 23].

Les modèles mentaux partagés aident les membres de l'équipe à mieux comprendre les intentions, les attentes et les besoins des autres membres, ce qui facilite la communication et la collaboration[22]. Par exemple, dans les équipes de développement logiciel, un modèle mental partagé du système en cours de construction peut aider les développeurs à comprendre comment

leurs contributions s'intègrent dans l'ensemble du projet, évitant ainsi les erreurs de conception et les problèmes de compatibilité[19].

De plus, les modèles mentaux partagés améliorent la coordination au sein des équipes en favorisant une compréhension commune des rôles et responsabilités de chaque membre, ainsi que des procédures et des processus à suivre[14]. Cela permet aux équipes de travailler ensemble de manière plus efficace et d'atteindre leurs objectifs plus rapidement.

Enfin, les modèles mentaux partagés facilitent la prise de décision en équipe en fournissant un cadre commun pour évaluer les alternatives et en aidant les membres de l'équipe à anticiper les conséquences de leurs actions[23]. Par exemple, dans les équipes de gestion de projet, un modèle mental partagé des objectifs et des contraintes du projet peut aider à identifier les risques et à élaborer des plans de contingence appropriés.

Il est néanmoins important de noter que les modèles mentaux partagés ne garantissent pas nécessairement une performance élevée de l'équipe. Des facteurs tels que le leadership, la motivation, la confiance et la cohésion de l'équipe peuvent également influencer la performance de l'équipe[24]. Par conséquent, il est essentiel d'examiner l'impact des modèles mentaux partagés sur la performance de l'équipe dans le contexte d'autres facteurs contribuant à la performance globale de l'équipe.

2.2 Méthode agile

En génie logicielle, les méthodes agiles servent à mettre en avant la relation entre une équipe auto-gérée et le client. Plus souple et flexible que les méthodologies dites traditionnelles telles que le *Waterfall*, l'Agile veut mettre au centre du projet les besoins du client[25].

2.2.1 Origine

Le mouvement Agile est né aux États-Unis, en février 2001, suite au constat du taux d'échec de plus en plus important des méthodes de gestion de projet classiques dans les années 1990[26]. Dix-sept experts désireux d'alléger les processus liés à la gestion de projet et au développement logiciel se réunissent afin de débattre de nouvelles pratiques et de mettre en avant des valeurs communes[27].

Suite à cette réflexion est né le terme Agile, associé à quatre valeurs fondamentales inscrites dans le manifeste pour le développement Agile de logiciels[28] :

— L'équipe, soit des individus et des interactions, plutôt que des processus et des outils ;
— L'application, c'est-à-dire des fonctionnalités opérationnelles plutôt que de la documentation exhaustive ;
— La collaboration avec le client, plutôt que la contractualisation des relations ;
— L'acceptation du changement, plutôt que le suivi d'un plan.

, ainsi qu'à douze principes :

— Satisfaire la clientèle en priorité
— Accueillir favorablement les demandes de changement
— Livrer le plus souvent possible des versions opérationnelles de l'application
— Assurer une coopération permanente entre le client et l'équipe projet
— Construire autour de personnes motivées
— Privilégier la conversation en face-à-face

— Mesurer l'avancement du projet en matière de fonctionnalité de l'application
— Faire avancer le projet à un rythme soutenable et constant
— Porter une attention continue à l'excellence technique et à la conception
— Faire simple
— Responsabiliser les équipes
— Ajuster à intervalles réguliers son comportement et ses processus pour être plus efficace

À noter que les deux méthodes Agile les plus utilisées étaient antérieures à cet événement. En effet, les fondements de Scrum étaient présentés en 1995 lors de la conférence OOPSLA[29] et la méthode Extreme Programming a été lancé officiellement en 1999[6].

Dans le cadre de ce mémoire, nous nous attarderons principalement sur la méthodologie Scrum étant donné qu'il s'agit de celle présente dans l'entreprise constituant notre étude de terrain.

2.2.2 Fonctionnement

Les méthodes agiles sont une approche de gestion de projet et de développement de logiciels basée sur un ensemble de valeurs et de principes qui privilégient la collaboration, la flexibilité et l'amélioration continue. Cette approche commence par la création d'équipes auto-organisées, composées de membres multidisciplinaires qui collaborent étroitement pour accomplir les tâches et atteindre les objectifs du projet.

Le travail est ensuite divisé en petites parties, appelées itérations ou sprints, qui durent généralement de deux à quatre semaines. À chaque itération, l'équipe travaille pour livrer une version fonctionnelle du produit ou du logiciel. La planification adaptative est un élément clé des méthodes agiles, ce qui signifie que la planification est flexible et évolutive. Les priorités et les objectifs peuvent changer au cours du projet en fonction des besoins des clients ou des conditions du marché.

La livraison incrémentielle est un autre aspect important des méthodes agiles. Le produit ou le logiciel est développé et livré par étapes, chaque itération ajoutant de nouvelles fonctionnalités et améliorations. Cela permet aux clients de bénéficier rapidement des avancées et d'apporter des commentaires précieux tout au long du processus de développement (Figure 14).

2.2.3 Rôles

La méthode agile Scrum repose sur trois rôles clés pour assurer le succès d'un projet : le Product Owner, le Scrum Master et l'équipe de développement. Chacun de ces rôles a des responsabilités distinctes et complémentaires.

Product Owner : Le Product Owner peut être un membre de l'équipe ou un représentant du client, et est généralement un expert du domaine métier concerné. Il est responsable de la gestion du Product Backlog, c'est-à-dire de la liste priorisée des fonctionnalités à développer. Le Product Owner travaille en étroite collaboration avec l'équipe de développement pour définir et clarifier les exigences, et s'assure que les objectifs du projet sont atteints.

Scrum Master : Le Scrum Master est un membre de l'équipe qui veille au bon déroulement du processus Scrum. Il facilite la communication et la collaboration entre le Product Owner et l'équipe de développement, et aide à résoudre les problèmes et les obstacles rencontrés en cours de projet. Le Scrum Master est également responsable de la promotion et du maintien des bonnes pratiques Scrum au sein de l'équipe.

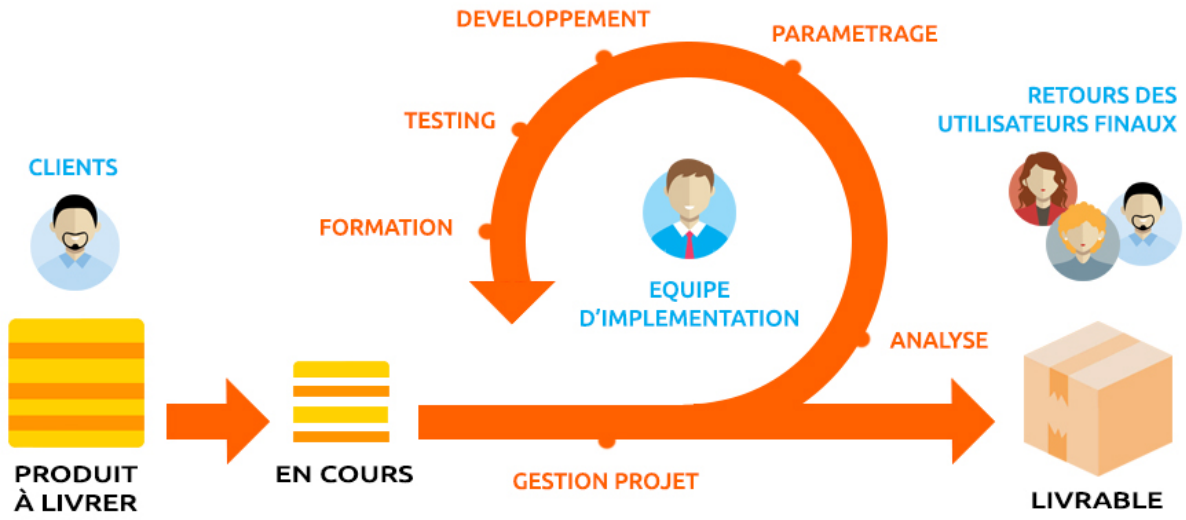

FIGURE 1 – Illustration du fonctionnement de la méthode agile Scrum [30]

Équipe de Développement : L'équipe de développement est composée de développeurs, analystes et testeurs qui travaillent ensemble pour concevoir, développer et tester les fonctionnalités du produit. L'équipe est auto-organisée et collabore étroitement pour planifier et exécuter les sprints, en se répartissant les tâches en fonction des compétences et des disponibilités de chacun.

La principale caractéristique d'une équipe Scrum est son auto-organisation, avec une absence de hiérarchie, y compris vis-à-vis du Scrum Master. L'ensemble de l'équipe est responsable de la réalisation du projet, sans distinction de rôles ou de compétences individuelles.

L'équipe participe également activement à la gestion du Sprint Backlog, en choisissant et en estimant les tâches à accomplir lors de chaque sprint.

2.2.4 Artefacts

Les artefacts Scrum sont des éléments clés qui aident l'équipe Scrum à organiser et suivre le travail tout au long du projet. Il y a trois artefacts principaux dans le cadre Scrum : le Product Backlog, le Sprint Backlog et l'Incrément de Produit.

Le Product Backlog : Le Product Backlog est une liste dynamique et priorisée de tous les éléments à inclure dans le produit final, tels que les fonctionnalités, les améliorations, les corrections de bugs et autres exigences. Chaque élément du Product Backlog est appelé User Story. Bien que chaque membre de l'équipe puisse consulter le Product Backlog, le Product Owner est responsable de sa maintenance, de sa priorisation et de son évolution. Il est important de noter que le Product Backlog est constamment mis à jour et ajusté en fonction des besoins du projet.

Le Sprint Backlog : Le Sprint Backlog est une liste de tâches sélectionnées à partir du Product Backlog, que l'équipe s'engage à accomplir au cours d'un sprint spécifique. Le Product Owner définit et priorise les éléments du Sprint Backlog en fonction des objectifs du sprint. Une fois que le sprint est lancé, l'équipe de développement travaille pour achever toutes les tâches du Sprint Backlog avant la fin du sprint.

L'Incrément de Produit : L'Incrément de Produit est le résultat du travail accompli pendant un sprint et représente la somme des éléments du Product Backlog terminés au cours

de ce sprint, ainsi que tous les incréments précédents. Cet incrément permet de montrer la progression du projet et de se rapprocher du produit final. L'Incrément de Produit doit être dans un état "potentiellement livrable", c'est-à-dire qu'il doit répondre aux critères de qualité définis par l'équipe et être prêt à être utilisé par les utilisateurs finaux si nécessaire.

2.3 Domain Driven Design

Le Domain Driven Design (DDD) est une approche de conception logicielle décrite par Eric Evans dans son ouvrage "Domain-driven design : tackling complexity in the heart of software"[31]. Il s'agit d'un ensemble de règles et de bonnes pratiques fondées sur le bon sens et qui souhaitent remettre au centre du développement logiciel le domaine métier.

2.3.1 Fonctionnement

L'objectif du DDD est de réduire la complexité requise lors du développement d'une solution en réalisant une implémentation aussi proche que possible du Domaine Métier du problème qu'elle doit résoudre.

Dans le cadre du DDD, les développeurs travaillent de façon proche avec les experts du Domaine qui sont en mesure d'expliquer comment le système fonctionne. De la sorte, ils construisent ensemble un langage (Ubiquitous Language) qui est compréhensible à la fois par les développeurs et les experts du Domaine.

Cet Ubiquitous Language est ensuite utilisé au sein du code lui-même.

Le but ultime du DDD est d'aider à construire des systèmes complexes qui sont plus faciles à maintenir, sont plus compréhensibles et permettent de répondre aux fonctionnalités demandées par le business, en modélisant le système sur base du Domaine lui-même : le design est le code et le code est le design.

Le DDD s'oppose la plupart du temps à une approche Anemic Domain Model (ADM) qui oblige à construire un modèle du Domaine, mais n'ajoute pas les rôles et responsabilités des objets de ce Domaine. Dans le cas de l'approche ADM, le modèle de données obtenu n'est qu'une projection d'un modèle relationnel entre les objets.

Lorsque l'on modélise un Domaine très large, il est progressivement de plus en plus difficile de construire un modèle unifié. Différents groupes de personnes utilisent des termes de vocabulaire subtilement différents dans différentes parties de l'organisation.

Prenons l'exemple d'une compagnie d'assurance. Avec le DDD, les développeurs et les experts métier (agents d'assurance, gestionnaires de sinistres, etc.) collaborent pour créer un langage ubiquitaire, comprenant des termes comme "Police", "Assuré", "Prime" ou "Sinistre".

L'Ubiquitous Langage se reflète dans le code, où chaque concept devient un objet ayant des responsabilités précises. Par exemple, un objet "Police" peut "calculer la prime" tandis qu'un objet "Assuré" peut "déclarer un sinistre". Ces objets sont conçus en utilisant Tactical Patterns du DDD comme les entités, les valeurs d'objet et les agrégats.

Cette approche facilite la compréhension du code et sa maintenance. Lorsque le domaine évolue (par exemple, une nouvelle règle sur le calcul des primes), seuls les objets et méthodes concernés doivent être modifiés. Ainsi, le DDD aide à gérer la complexité du développement logiciel dans le contexte spécifique de l'assurance.

2.3.2 Ubiquitous Language et Bounded Context

Dès lors, à la place de ce modèle unifié, le DDD divise ce dernier en Bounded Contexts, possédant chacun un modèle unifié en leur sein. Chaque Bounded Context utilise son propre Ubiquitous Language.

Les Bounded Contexts possèdent à la fois des concepts distincts des autres Bounded Contexts (par exemple, un ticket de support existe uniquement dans un contexte de support client) mais également des concepts communs (comme des produits et des clients).

Plusieurs Bounded Contexts peuvent avoir des modélisations complètement différentes de concepts communs, avec des mécanismes d'intégration mappant ces concepts polysémiques (qui possèdent plusieurs sens) entre les Bounded Contexts.

2.3.3 Tactical Pattern

Les modèles tactiques du DDD aident à mettre en œuvre concrètement le modèle conceptuel dans le code. Ils regroupent une série de patterns de conception, qui permettent de modéliser le domaine avec précision et flexibilité.

Ces patterns incluent les Entités, les Valeurs d'Objet, les Agrégats, les Événements de Domaine, les Services de Domaine, etc. Chacun a un rôle spécifique et répond à des besoins particuliers.

Par exemple, une Entité a une identité qui la rend unique, tandis qu'une Valeur d'Objet est définie uniquement par ses attributs. Un Agrégat est un cluster d'Entités et de Valeurs d'Objet qui peuvent être traitées comme une unité unique pour garantir la cohérence du domaine.

Les modèles tactiques offrent une façon cohérente et prévisible de concevoir et d'implémenter le code, facilitant la maintenance et l'évolution du système. Ils reflètent les concepts et opérations du domaine, tels que définis dans le langage ubiquitaire et structurés dans les Bounded Contexts."

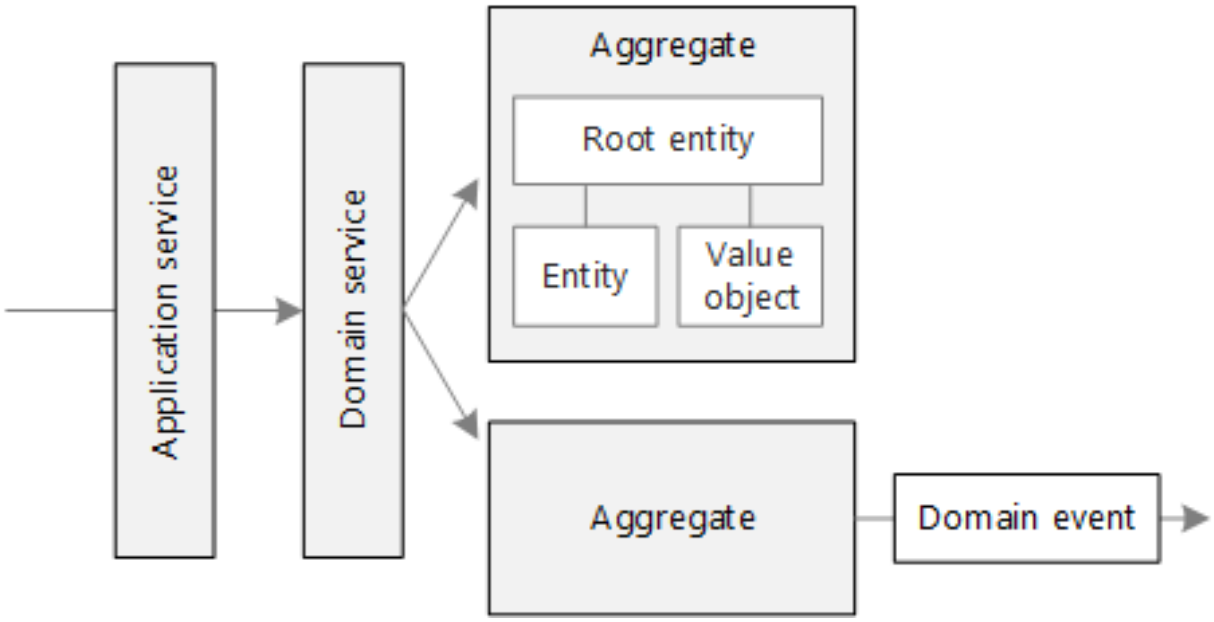

Figure 2 – Overview[32]

2.3.4 Évènements asynchrones

L'utilisation d'événements asynchrones en DDD peut apporter plusieurs avantages, notamment :

— Découplage des composants : Les événements asynchrones permettent de découpler les différentes parties de l'application. Plutôt que de dépendre directement les uns des autres, les composants peuvent communiquer via des événements. Cela permet de simplifier l'architecture et de faciliter la maintenance de l'application.

— Faciliter la mise à l'échelle : En utilisant des événements asynchrones, il est plus facile de mettre à l'échelle certaines parties de l'application, car chaque composant peut être exécuté indépendamment des autres. Cela permet de mieux gérer les pics de trafic et d'augmenter la résilience de l'application.

— Gestion de la complexité : Les applications basées sur le DDD peuvent être complexes, avec de nombreux domaines et Entities interdépendantes. En utilisant des événements asynchrones, il est possible de simplifier la gestion de cette complexité en permettant aux différents domaines de communiquer de manière asynchrone.

— Traçabilité : Les événements asynchrones peuvent être utilisés pour tracer l'activité de l'application. Par exemple, en enregistrant tous les événements qui se produisent dans l'application, il est possible de remonter à l'origine de certains problèmes et de mieux comprendre le comportement de l'application.

Néanmoins, les évènements asynchrones peuvent également générer quelques difficultés aux équipes de développement :

— La synchronisation de l'état : Lorsqu'un événement est déclenché, il peut prendre un certain temps avant que l'application réagisse. Cela peut entraîner des problèmes de synchronisation de l'état entre les différentes parties de l'application.

— La gestion des conflits : Dans un système distribué, il peut arriver que plusieurs événements soient déclenchés en même temps, menant potentiellement à des conflits. Par exemple, si deux événements tentent simultanément de modifier une même ressource, on peut se retrouver face à ce qu'on appelle une "condition de course". Cette situation se produit quand l'ordre d'exécution des opérations n'est pas contrôlé et conduit à un état indésirable. En effet, la ressource peut se retrouver dans un état inattendu ou certaines modifications peuvent être perdues. Il est donc crucial d'implémenter des mécanismes de contrôle pour prévenir ces conditions de course et assurer l'intégrité de la ressource.

— La complexité du modèle de domaine : Le modèle de domaine dans le DDD peut être complexe, avec de nombreuses Entities et relations. La gestion des événements asynchrones doit donc prendre en compte cette complexité.

— La tolérance aux pannes : La tolérance aux pannes est un élément clé de tout système distribué. La gestion des événements asynchrones doit donc prendre en compte les pannes éventuelles, et être en mesure de récupérer les événements perdus ou non traités.

En résumé, l'utilisation d'événements asynchrones en DDD peut faciliter la mise en œuvre de l'architecture, faciliter la mise à l'échelle, simplifier la gestion de la complexité et offrir des avantages en termes de traçabilité.

2.3.5 Avantages du DDD

Les avantages du DDD sont les suivants :

— L'organisation gagne un modèle utile de son Domaine
— Une définition raffinée, précise et compréhensible du business de l'organisation est développée
— Les experts du domaine contribuent au design du logiciel
— Une meilleure expérience utilisateur
— Des frontières claires sont placées entre les modèles (Bounded Contexts)
— L'architecture de l'entreprise est mieux organisée
— Une modélisation Agile, interactive et continue est utilisée
— De nouveaux outils, tant stratégiques que tactiques, sont utilisés

2.4 Une approche conjointe

Le DDD et les méthodes agiles se complètent efficacement, étant donné qu'elles sont basées sur des valeurs et des principes similaires.

Tout d'abord, les approches Agile et DDD mettent tous les deux l'accent sur la collaboration entre les développeurs et les experts métier pour garantir que les modèles de données reflètent correctement les besoins de l'entreprise. Cette collaboration est essentielle pour garantir que les logiciels créés répondent aux besoins réels de l'entreprise.

Ensuite, les deux approches mettent l'accent sur l'adaptabilité et l'évolution constante. Le DDD encourage une approche évolutionniste pour la création de modèles de données, en permettant aux modèles de s'adapter aux changements dans les besoins métier. De même, les méthodes agiles promeuvent une approche itérative et incrémentale pour le développement de logiciels, en permettant aux équipes de s'adapter aux changements dans les besoins métier.

Enfin, les Bounded Contexts du DDD sont en adéquation avec le principe de l'empirisme des méthodes agiles qui repose sur une boucle d'apprentissage en continu et une adaptation constante. Les Bounded Contexts permettent d'isoler les différents modèles de données de l'entreprise en les isolant dans des contextes définis, ce qui permet aux équipes de s'adapter aux changements dans les besoins métier sans affecter les autres parties du système.

En somme, les approches Agile et DDD sont complémentaires, car elles mettent tous les deux l'accent sur la collaboration, l'adaptabilité et l'évolution constante, ce qui permet de créer des logiciels qui répondent aux besoins réels de l'entreprise de manière efficace.

Il existe plusieurs façons de combiner les méthodes agiles et le DDD.

— Utilisation de sprints DDD : Les sprints DDD sont une extension des sprints agiles traditionnels, se concentrant sur la compréhension des besoins métier et la création de modèles de données qui reflètent ces besoins. Les activités des sprints DDD comprennent la définition des Aggregates (groupes de choses qui peuvent être traitées ensemble), l'identification des Entities (objets uniques dans le système) et la création de Services (opérations qui n'appartiennent pas naturellement à un objet ou une entité).
— Utilisation de l'Event Storming : L'Event Storming est une technique de modélisation collaborative qui permet aux équipes agile de comprendre les besoins métier de l'entreprise en utilisant des événements métier pour créer un modèle de données. Cette technique peut être utilisée pour identifier les Aggregates, les Entities et les Services nécessaires pour un système.

— Utilisation des User Story Mapping : Les User Story Mapping sont un outil de planification agile qui peut être utilisé pour comprendre les besoins métier de l'entreprise. Les User Story Mapping peuvent être utilisés pour identifier les Aggregates, les Entities et les Services nécessaires pour un système, ainsi que pour définir les relations entre ces éléments.

— Utilisation d'un backlog DDD : Le backlog DDD est une extension du backlog agile traditionnel qui se concentre sur les besoins métier de l'entreprise. Il peut inclure des éléments tels que les Aggregates, les Entities et les Services, ainsi que des exigences de domaine spécifiques qui doivent être prises en compte lors de la création d'un système.

En utilisant ces outils et techniques, les équipes Agile peuvent mieux comprendre les besoins métier de l'entreprise et créer des modèles de données qui reflètent ces besoins. Cela permet de créer des logiciels qui répondent aux besoins réels de l'entreprise de manière efficace, en combinant les avantages de l'Agile et de DDD.

3 État de l'art

Dans le cadre de notre travail, nous avons réalisé une revue littéraire non exhaustive reprenant les points centraux de notre recherche, à savoir l'étude des modèles mentaux partagés, mais appliqués distinctement au développement logiciel dans son ensemble, à l'application des méthodes agiles ou encore en appliquant le DDD. Par la suite, nous avons recensé et résumé les points les plus récurrents afin de proposer une vue d'ensemble sur le sujet.

3.1 Résultat des recherches

Les modèles mentaux partagés sont définis par Cannon-Bowers & Salas [33] comme les structures de connaissances détenues par les membres d'une équipe qui leur permettent de former des explications et des attentes précises pour la tâche, et à leur tour, de coordonner leurs actions et d'adapter leur comportement aux exigences de la tâche et des autres membres de l'équipe.

La théorie des modèles mentaux partagés étend l'idée du modèle mental d'un individu pour conceptualiser les équipes comme une unité de traitement de l'information unifiée. Un modèle mental partagé est défini comme "les structures de connaissances détenues par les membres d'une équipe qui leur permettent de former des explications et des attentes précises pour la tâche, et, à leur tour, de coordonner leurs actions et d'adapter leur comportement aux exigences de la tâche et des autres membres de l'équipe"[33]. Les modèles mentaux partagés fournissent à l'équipe une base de connaissances interne qui permet aux membres de l'équipe de décider des actions à entreprendre lorsque des événements nouveaux se produisent. Grâce à eux, les équipes forment des attentes compatibles de la tâche et de l'équipe. Ces modèles mentaux partagés aident l'équipe à comprendre les événements actuels, à envisager ce qui pourrait se produire dans un avenir proche pour le groupe et à comprendre pourquoi les événements se produisent[34].

3.2 Perception générale des modèles mentaux partagés

Afin de donner un point de vue aussi complet que possible de la notion de modèle mentaux partagé dans une équipe de développement logiciel, nous avons tenté de catégoriser les points d'attention récurrents des diverses études de notre état de l'art. Les cinq sections qui suivent se concentrent chacun sur un aspect différent de ce thème.

La première section traite des développeurs et de la production de code. Il souligne l'importance de la compréhension mutuelle et du partage de connaissances entre les membres d'une équipe de développement pour produire un code de qualité et pour éviter les problèmes liés à une mauvaise communication.

La deuxième section se concentre sur les compétences cognitives nécessaires pour développer des modèles mentaux partagés efficaces. Il explore les différents types de compétences cognitives et comment les développeurs peuvent les développer pour améliorer leur compréhension mutuelle.

La troisième section évoque un contre-exemple géographique pour illustrer comment les modèles mentaux partagés peuvent être affectés par des différences culturelles et linguistiques. Il souligne l'importance de comprendre ces différences pour faciliter une communication efficace et pour éviter les malentendus.

La quatrième section se concentre sur le développement asynchrone et sur la manière dont les outils de communication peuvent être utilisés pour faciliter la compréhension mutuelle et le

partage de connaissances entre les membres d'une équipe de développement qui travaillent à distance.

Enfin, la dernière section traite de l'architecture logicielle et de la manière dont elle peut être utilisée pour faciliter la communication et la compréhension mutuelle entre les membres d'une équipe de développement. Il explore les différents principes de l'architecture logicielle qui peuvent aider à développer des modèles mentaux partagés efficaces, tels que la loi de l'ignorance maximale, la loi du changement minimal, la loi de la permanence de la position et le principe de l'esthétique.

Néanmoins, nous n'avons pas l'ambition de réaliser un état de l'art totalement exhaustif, tant le sujet est varié, mais nous espérons en avoir tiré les grandes lignes directives.

3.2.1 Développeurs et production de code

Les membres de l'équipe, et plus particulièrement les développeurs, doivent connaître ou obtenir une variété d'informations pour réussir à comprendre et éditer un code dont ils ne sont pas les initiateurs, comment les décisions de conception sont représentées dans le code, la logique derrière celles-ci, le propriétaire responsable du code, les autres développeurs qui travaillent actuellement dessus, les changements qui risquent d'impacter le code ailleurs, et ceux qui ont des répercutions sur celui-ci.

Les développeurs ont de nombreux moyens pour enregistrer, communiquer et découvrir ces informations. Les conventions de nommage, les commentaires et les documents d'architecture permettent aux développeurs de partager leur compréhension actuelle avec les futurs développeurs, mais nécessitent aussi un investissement en temps et en connaissances aussi bien pour ceux qui les rédigent que pour les personnes qu'elles ciblent. Les conventions, la factorisation et les patterns de conception minimisent la charge de la documentation en fournissant des réponses générales, mais limitent les solutions possibles et deviennent eux-mêmes plus à apprendre.

Lavazza et al[35] mettent d'ailleurs en avant les moyens de communication, directs ou indirects, employés par les développeurs pour faciliter ce passage de témoin sur le code et les problèmes qui peuvent en découler ainsi que sur les modèles mentaux du code lui-même.

Les développeurs créent et maintiennent des modes mentaux complexes du code. Suite aux entretiens de l'étude précitée, nous savons que les développeurs, sans se référer à des documents écrits, peuvent parler en détail de l'architecture de leur projet, de conception de celle-ci, de la propriété des parties du code, de l'histoire de celui-ci, des tâches à accomplir ou toutes informations relatives à la bonne compréhension du code. Dans la plupart des cas, ces connaissances ne sont jamais ou rarement écrites, sauf sous des formes transitoires comme des croquis sur un tableau blanc. On peut donc en déduire que la connaissance relative au code se retrouve la plupart du temps dans la tête de celui qui l'a écrit ou qui travaille actuellement dessus.

Les modèles mentaux sont coûteux à créer et à maintenir. En effet, les développeurs ont une forte notion de propriété personnelle du code, ce qui limite la quantité qu'ils doivent comprendre en détail. D'autre part, certaines équipes ont pour politique d'éviter la propriété personnelle du code, car elle rend les individus trop indispensables et favorise "trop de passion autour du code" selon l'une des personnes interrogée.

On peut également parler de notion de propriété du code au sein de l'équipe, cette notion étant d'ailleurs encore plus forte et présente que celle de la propriété du développeur. Les équipes de développement travaillent presque toujours dans un même endroit, ce qui participe au partage informel des connaissances. Un autre moyen utilisé par les développeurs pour maintenir

leur modèle mental du code de leur équipe est de s'abonner aux notifications par email de logiciel tels que *Jira* ou *Confluence*, même s'il apparaît que ces notifications manquent parfois de contexte ou de détails quant à la raison de leur création.

Créer un modèle mental à partir de rien demande beaucoup d'énergie au nouveau membre et à l'équipe dans son ensemble. Le nouveau venu se voit souvent attribuer un mentor, souvent un membre ancien de l'équipe, désigné comme premier point de contact pour les questions relatives au code. Les nouveaux arrivants sont beaucoup plus susceptibles de lire les documents de conception de l'équipe que les membres expérimentés. Certaines équipes mettent aussi en ligne des documents spécialement destinés aux nouveaux arrivants.

Enfin l'étude évoque la problématique de la duplication de code. En effet même si celle-ci est une pratique avouée et couramment utilisée dans le développement logiciel, elle n'est pas toujours sans conséquence pour l'évolution des modèles mentaux de l'équipe et la crédibilité du projet. Six formes distinctes de duplication de code ont été détectées, chacune d'elle pouvant se définir par son mécanisme de création, le fait que les développeurs soient conscients ou non qu'ils créent des doublons et les défis de refactoring pour les supprimer.

— le doublon répété : quand deux développeurs travaillent sur la même implémention sans le savoir ;

— le doublon d'exemple : quand un exemple sorti d'une documentation officielle est copié/-collé ; puis modifié pour être intégré au projet. On peut considérer qu'il s'agit en général de petites parties de code ;

— le doublon dispersé : il s'agit d'une modification du code en entrainant d'autres, plus petites, à divers endroits du projet ;

— le doublon de fourche : quand une équipe duplique le code d'une autre équipe si celui-ci n'est pas encore livré ou totalement supporté ;

— le doublon de branche : il s'agit de dupliquer du code dans plusieurs branches d'un même projet comme aligner la production avec le développement ;

— le doublon de langage : quand on implémente la même méthode dans plusieurs langages différents.

Cette problématique est principalement due au manque de documentation formelle du code et de la conception même du projet. De manière générale les développeurs déclarent d'abord essayer de comprendre le code par eux-mêmes, en lisant les commentaires, invariants et spécifications, avant de rechercher une éventuelle documentation ou d'interroger leurs collègues.

L'étude de Latozza et al[35] met en avant l'importance de la nature profondément sociale du développement logiciel, la communication inter-équipe et la manière dont peuvent évoluer les modèles mentaux partagés dans ce contexte.

3.2.2 Modèles mentaux et compétences cognitives

Dans la section précédente, nous avons discuté des moyens pour les développeurs de comprendre et de communiquer efficacement sur le code. Nous avons évoqué les modèles mentaux complexes que les développeurs créent pour comprendre le code et les moyens de les maintenir au sein de l'équipe. Cependant, la création et la maintenance de ces modèles mentaux sont coûteuses et peuvent être difficiles pour les nouveaux membres de l'équipe. Dans cette section, nous explorerons les compétences cognitives nécessaires pour les développeurs afin de créer et de maintenir ces modèles mentaux. Nous examinerons également les moyens d'améliorer ces compétences pour faciliter la compréhension et la communication du code.

Yang et al[36] basent leur réflexion à propos des modèles mentaux sur 3 axes de recherche : deux compétences cognitives que sont les compétences en forme de A et celles en forme de T ainsi que l'agréabilité des membres de l'équipe.

Le concept de gestion des compétences en T est une métaphore utilisée pour décrire l'évolution des capacités des personnes. La barre verticale du T représente le domaine d'expertise d'une personne dans un sujet précis tandis que la barre horizontale représente sa capacité à appliquer son domaine d'expertise dans des domaines connexes au sien. Par conséquent, elle peut donc être définie comme la capacité de combiner ou d'intégrer les compétences ou les connaissances des autres à sa propre base de connaissances. Il a été affirmé que les personnes polyvalentes sont plus susceptibles d'être en mesure d'assimiler diverses connaissances et compétences. On peut s'attendre à ce qu'une telle capacité stimule les conflits créatifs et promulgue une communication active entre les membres de l'équipe[37]. Par conséquent, Yang et al[36] affirment que les personnes ayant des compétences en forme de T au sein des équipes de développement facilitent l'établissement des modèles mentaux partagés de celle-ci.

À l'instar des compétences en forme de T, les compétences en forme de A servent également à décrire les capacités d'une personne. En outre, elles suggèrent plutôt que les individus ont tendance à former une alliance technique de plusieurs disciplines afin d'augmenter leur compréhension à un niveau d'abstraction plus élevé. La compétence en forme de "A dénote la capacité individuelle à intégrer différentes disciplines et de les greffer en une seule. Madhavan & Grover[38] ont proposé que les compétences en forme de A des membres de l'équipe sont positivement liées avec les performances de l'équipe en matière de création de connaissances. Grâce au développement d'un ensemble de compétences en forme de A sur l'individu et/ou l'organisation, il n'est peut-être pas nécessaire de s'assurer que tous ses membres possèdent ces connaissances. Ils affirment qu'il s'agit d'un calibre critique pour les chefs d'équipe afin d'intégrer diverses connaissances et en développer de nouvelles, et que les équipes sont meilleures avec des leaders ayant un tel calibre que les équipes sans. Par conséquent, Yang et al[36] estiment que l'intégration de plusieurs profils en forme de A dans une équipe améliore les modèles mentaux partagés de celle-ci.

Enfin l'étude évoque également des aspects non cognitifs de l'équipe et plus particulièrement de l'agréabilité des membres de celle-ci. L'agréabilité consiste à maintenir une relation de confiance et de coopération avec les autres. Les personnes ayant un score élevé d'agréabilité sont courtoises, coopératives, généreuses, conciliantes, prévenantes, patientes et chaleureuses, alors que les personnes ayant un profil bas sont plus sensibles à la violence, indifférentes aux autres, égocentriques et jalouses[39]. L'agréabilité favorise la cohésion du groupe et contribue à la performance de l'équipe. L'étude rapporte également que les coéquipiers ayant un score élevé d'agréabilité ont tendance à avoir une grande sympathie, à coopérer facilement avec les autres membres de l'équipe et sont volontaires pour aider les autres, de sorte qu'ils facilitent la cohésion de l'équipe en favorisant la convivialité entre eux. Par conséquent l'étude suggère que les membres d'une équipe ayant un score élevé d'agréabilité ne sont pas exclusifs quant à l'opinion des autres et coopèrent pour développer le modèle mental partagé sur les procédures de l'équipe et l'attribution des rôles.

3.2.3 Contre exemple

Cependant, il arrive que les hypothèses formulées ci-dessus, comme quoi le travail d'équipe permet aux modèles mentaux partagés de devenir plus similaire au fil du temps, ne se confirment pas. Levesque et al[40] émettent l'hypothèse que l'interaction au sein de l'équipe permet aux modèles mentaux partagés d'évoluer de manière favorable pendant la durée de vie du projet,

peu importe les membres qui composent celui-ci. Plus les membres du groupe communiquent entre eux, plus ils sont susceptibles de former un cadre de référence commun et de développer un modèle mental partagé entre eux[7]. L'un des critères fondamentaux pour que l'interaction soit efficace au sein de l'équipe est la structure des tâches ou le degré de différenciation des rôles. Les membres de l'équipe communiquent différemment selon la façon dont leurs rôles sont structurés.

Puisque l'interaction du groupe pour coordonner le travail tient en partie compte de la répartition des tâches au sein du groupe, il peut y avoir des différences entre les équipes en son sein. Ainsi, certaines situations peuvent être moins propices à la formation de modèles mentaux partagés, comme la distribution géographique de ses membres.

Bien que la communication et la structure des tâches soient des éléments clés pour favoriser la formation de modèles mentaux partagés au sein d'une équipe, cela ne garantit pas toujours leur succès. En effet, il peut y avoir des situations où la distribution géographique des membres ou la spécialisation croissante des rôles peuvent entraver le partage des connaissances et la communication au sein du groupe. Cela a été mis en évidence par l'étude de Levesque et al[40], qui a constaté une diminution des modèles mentaux partagés au fil du temps chez certaines équipes de développement logiciel. Ainsi, pour comprendre pleinement la dynamique des modèles mentaux partagés au sein des équipes, il est important d'examiner les variables du groupe dans le temps et l'espace.

Dans le cas de ces équipes, la diminution des modèles mentaux partagés était liée à une diminution de l'interaction dans ces groupes. Au cours du projet de développement logiciel, les rôles des membres du groupe sont devenus de plus en plus spécialisés, et le temps passé par les membres à travailler ou à communiquer les uns avec les autres a diminué en conséquence. Cette étude souligne l'importance d'examiner les variables du groupe dans le temps et l'espace, de même qu'elle indique clairement que ces indicateurs comptent dans le développement des modèles mentaux partagés.

3.2.4 Impact de la distance et du temps

Le développement asynchrone est un sujet crucial dans le monde du développement logiciel, notamment pour les projets open source et de grande envergure. Dans les chapitres précédents, nous avons vu les défis que les développeurs rencontrent lorsqu'ils travaillent à distance et comment les pratiques de communication, de coordination et de collaboration peuvent affecter le développement logiciel. Dans ce chapitre, nous allons examiner comment la distance et le temps peuvent impacter le développement asynchrone et les stratégies que les développeurs peuvent adopter pour surmonter ces obstacles. Nous allons notamment nous baser sur les études de Barbara Scozzi[41] et al. et Alberto Espinosa et al.[42] qui ont étudié l'impact de la distance et du temps sur le développement logiciel.

En effet, Barbara Scozzi et al[41] étudient les interactions d'une équipe de développement sur un projet *open source* et prennent l'hypothèse qu'il existe plusieurs aspects différents des modèles mentaux qui peuvent être partagés en leur sein malgré les difficultés qu'elles peuvent rencontrer. Les auteurs dégagent ainsi différentes catégories de pratiques :

— *Les schémas d'interprétation partagés* concernant les tâches et les capacités des acteurs peuvent permettre aux équipes de coordonner leurs activités sans avoir besoin de communications explicites. Le problème du développement de schémas d'interprétation partagés est susceptible d'affecter particulièrement le développement des projets *open source*, puisque les membres des équipes *open source* sont distribués, ont des origines diverses, et

rejoignent ces équipes à différentes phases du processus de développement logiciel. Ils permettent de guider les contributions individuelles efficaces et la coordination du processus de développement logiciel.

— *La compréhension des comportements attendus qui constitue un rôle* : des études antérieures ont décrit comment certains individus du groupe peuvent passer d'un rôle à l'autre en fonction de leur implication dans le projet. Selon les projets *open source*, ce transfert d'un rôle à l'autre peut se faire de manière totalement formel, par des votes au sein de la communauté, ou sur base informelle, géré par l'initiateur même du projet. Il est donc nécessaire, au vu de la nature du projet, de définir de manière claire le rôle des membres de l'équipe tout en prenant compte des nombreux participants qu'engendrent les développements *open source*. Même si l'on pourrait d'abord penser que ce nombre important est un frein au développement, celui-ci est justement repris pas ces membres comme la force des projets f*open source*.

— *Les règles et les normes* : Dans la vision de l'entreprise basée sur la connaissance, les groupes sont définis comme "des entités sociales qui détiennent des règles implicites et explicites qui guident l'interprétation, la contribution et le comportement d'un membre individuel"[43]. Dans les projets *open source* l'édiction de règles et normes est d'autant plus important si l'on désire conserver une cohérence du projet sur le long terme. Ces règles sont systématiquement consignées de manière formelle et conventionnelle par l'équipe et mises à disposition de l'ensemble de la communauté.

L'étude de Barbara Scozzi et al[41] montre que l'on peut atteindre un niveau de modèles mentaux partagés malgré les difficultés inhérentes au projet *open source* et aux origines diverses des membres de l'équipe.

Un autre point de vue concernant la problématique d'une large distribution des ressources sur le développement d'un projet de grande envergure est celui d'Alberto Espinosa et al[42].

La composition des membres de l'équipe est un des facteurs essentiels dans la réussite d'un projet de développement sur le long terme. En effet, les origines, expériences et tout ce qui constitue les modèles mentaux individuels des différents membres du projet influencent grandement le bagage de départ de l'équipe et, *de facto*, les modèles mentaux partagés. Le partage du modèle est d'autant plus important entre les principaux membres initiateurs du projet, au vus des personnes rejoignant l'équipe en cours de route et devant s'ancrer sur des bases solides[41].

Pour eux le développement de logiciels à grande échelle est une activité collaborative qui nécessite des compétences diverses et des ressources humaines importantes. Néanmoins, au fur et mesure que le projet prend de l'ampleur, il est souvent moins pratique voir difficile de concentrer toutes les ressources humaines nécessaires en un seul et même endroit. En outre, les outils avancés de télécommunication et de collaboration d'aujourd'hui constituent une incitation supplémentaire à collaborer avec des collègues géographiquement éloignés et dispersés. Cependant, malgré l'abondance d'outils de collaboration, il n'en demeure pas moins que la coordination dans le cadre du développement de logiciels distribués à grande échelle reste problématique pour de nombreuses organisations, car le travail à distance entraîne une augmentation des frais généraux de coordination, une communication moins efficiente et des retards plus importants[44].

Si nous savons déjà, grâce à nos précédentes lectures, que les modèles mentaux partagés sont efficaces au sein d'une même équipe, les collaborateurs asynchrones et distribués ont moins d'occasions d'interagir que les collaborateurs en temps réel et colocalisés, ce qui rend plus difficile le développement de modèles mentaux partagés.

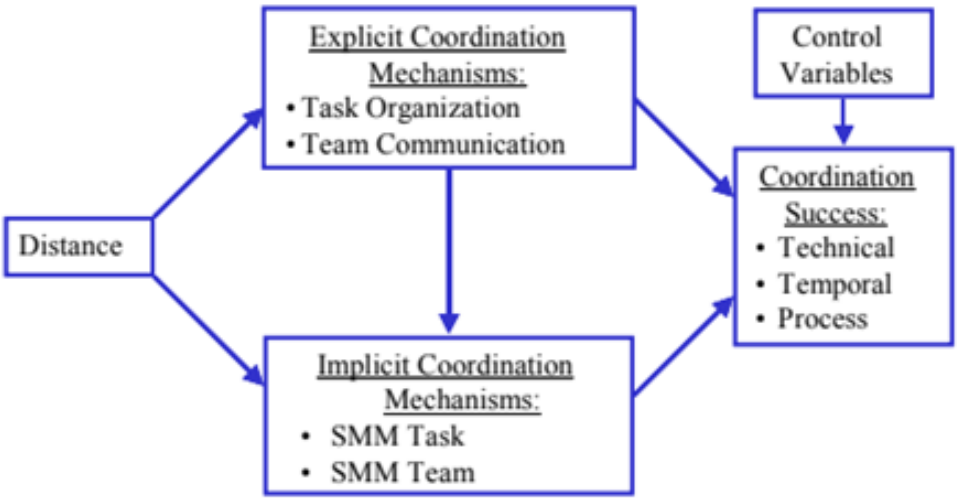

FIGURE 3 – Impact de la distance [42]

On sait qu'à mesure que les collaborateurs acquièrent de l'expérience avec la tâche et auprès des autres membres de l'équipe, ils développent également des modèles mentaux partagés plus rapidement et efficacement[23]. Les modèles mentaux partagés sont basés sur des connaissances partagées organisées, qui aident les collaborateurs à former des explications et des attentes précises sur la tâche, les aidant ainsi à se coordonner explicitement[33]. Dès lors une activité de développement asynchrone et des membres géographiquement distribués peuvent mener à des actions non coordonnées et à des pertes de productivité dues à des éléments tels que le *reworking* et le non-respect des délais, car les différentes dépendances entre les tâches peuvent ne pas être gérées de manière adéquate à cause du manque de communication entre les membres.

Le schéma ci-dessus résume les conclusions de Espinosa et al[42] : la distance entre les membres d'une même équipe influence à la fois les mécanismes de coordination explicites comme la communication et l'organisation, mais aussi implicite, à l'instar des modèles mentaux des tâches et de l'équipe. Enfin les variables de contrôle servant d'indicateur de réussite du projet tels que les aspects techniques (est-ce que le logiciel fonctionne comme il le devrait ?), temporel (le logiciel a-t-il été livré à la date prévue dans le calendrier du projet ?) et les processus même de développement (coordination des décision) sont directement impactées.

Une autre étude de Espinosa et al[45] explique la notion de distance dans les projets de développement et la difficulté qu'elle peut engendrer, mais se différencie par son approche de la notion de familiarité. En effet, malgré les distances géographiques ou de temps, il est néanmoins possible de construire des modèles mentaux partagés solides : la familiarité avec le travail fait référence aux connaissances que possèdent les membres individuels. Les modèles mentaux partagés par les membres de l'équipe se développent à partir du travail et de la formation en commun sur des tâches similaires[40] et à partir d'éléments tels que l'expérience de l'équipe et la familiarité organisationnelle commune[46]. Par conséquent, les membres d'une équipe qui a l'habitude de travailler sur des tâches similaires sont plus susceptibles d'avoir des modèles mentaux partagés plus forts malgré les éventuelles distances qui les séparent.

3.2.5 Architecture logicielle

Dans les chapitres précédents, nous avons exploré l'importance de l'architecture logicielle pour la qualité et la maintenance des logiciels. Nous avons vu que l'architecture logicielle permet de créer un plan de construction solide et durable pour le développement de logiciels. Dans ce chapitre, nous avons approfondi la conception de l'architecture logicielle en adoptant l'approche de Holt, qui affirme que l'architecture logicielle est un modèle mental partagé par les personnes responsables du logiciel. Nous avons vu que les principes de Holt[47] peuvent aider à créer un modèle mental partagé efficace pour les développeurs, ce qui peut faciliter la communication et la compréhension de l'équipe. En somme, la conception d'une architecture logicielle solide et durable est un élément clé de la création de logiciels de qualité et de l'optimisation du travail en équipe.

Communément considéré comme un ensemble de composants, structures ou encore protocoles donnant les grandes lignes directrices d'un projet[48], l'approche de Holt[47] adopte une position différente, affirmant que l'architecture logicielle est plus utilement considérée comme un modèle mental partagé par les personnes responsables du logiciel. Pour Holt ce modèle est considéré comme l'essence même de l'architecture logicielle.

Il divise les principes d'une bonne architecture dans les concepts cognitifs suivants :

— La loi de l'ignorance maximale : il n'est pas toujours bon, ni possible, de tout apprendre avant de commencer à travailler sur un projet. Dès lors une architecture légère et concise, basée sur un visuel fort aide à la construction de modèles mentaux partagés efficaces ;

— La loi du changement minimal : lorsque le logiciel évolue de manière modeste, le modèle que nous lui appliquons doit également évoluer de manière minimale. Une fois qu'une équipe a investi le temps nécessaire pour visualiser l'architecture de son système logiciel, elle ne doit pas modifier inutilement son modèle mental du système. Chaque changement de ce modèle dans dans la tête des développeurs prend du temps, provoque de la confusion et est source d'erreurs ;

— Loi de la permanence de la position : les visualisations des versions d'un système doivent montrer les parties correspondantes du système à peu près aux mêmes positions, avec à peu près les mêmes tailles et formes. Par exemple si dans un schéma d'activité les interactions avec la base de données sont situées en bas du diagramme, il est inutile de modifier cette position sans raison valable ;

— Le principe de l'esthétique : de manière naturelle l'individu aura toujours une préférence pour quelque chose d'aspect simple et propre. Dès lors, il aura tendance à ignorer ou réparer les aspects qui lui déplaisent. Ces principe concernant la laideur et la propreté de l'architecture expliquent pourquoi la visualisation des logiciels conduit à de meilleurs logiciels.

L'étude conclut en rappelant que l'architecture est intimement liée à la structure sociale de l'équipe de développement. Il faut se rappeler que l'architecture est utilisée en grande partie dans la tête des gens, pour réfléchir à ce qu'est l'architecture et comment la modifier, puis pour communiquer ces idées à d'autres personnes.

L'objectif principal de l'architecture logicielle est de faciliter la communication et la compréhension de l'équipe. Pour qu'une équipe puisse communiquer, elle a besoin d'un "vocabulaire" commun. Mais cette compréhension mutuelle doit être bien plus profonde que de simples éléments de langages partagés. La compréhension doit plutôt inclure un modèle partagé de ce que sont les parties du système et comment elles interagissent.

3.3 Modèles mentaux partagés et méthode agile

Le succès d'un projet de développement logiciel dépend fortement des performances de l'équipe, comme tout processus impliquant une interaction humaine. Une définition commune d'une équipe est " un petit nombre de personnes ayant des compétences complémentaires et qui sont engagées dans un but commun, un ensemble d'objectifs de performance et une approche pour lesquels ils se tiennent mutuellement responsables"[49].

D'un point de vue traditionnel, le développement logiciel se joint davantage à une approche rationaliste, axée sur les lignes de produit et fondée sur un plan en utilisant une approche normalisée, contrôlable et prévisible[50].

Aujourd'hui les méthodes agiles se différencient par leur approche basée sur la flexibilité et la réactivité, en opposition au contrôle et à la vérification des approches traditionnelles[51]. Le développement agile est une méthode de développement logiciel légère et incrémentale dont les pratiques spécifiques mettent l'accent sur une interaction étroite avec les clients. L'un des avantages des pratiques de développement logiciel agile est la capacité des équipes de développement logiciel à s'adapter aux exigences changeantes des clients, tout en identifiant et en réduisant certains risques qui surviennent au cours du développement logiciel[52].

La notion de modèles mentaux partagés est essentielle dans le développement logiciel agile, car elle permet à l'équipe de travailler efficacement ensemble. Comme mentionné précédemment, une équipe de développement logiciel est composée d'un petit nombre de personnes ayant des compétences complémentaires et qui sont engagées dans un but commun. L'équipe agile, en particulier, doit être flexible et réactive, afin de s'adapter aux exigences changeantes des clients.

Les modèles mentaux partagés sont des représentations mentales partagées par les membres de l'équipe sur les objectifs du projet, les rôles et les responsabilités de chacun, ainsi que sur les défis techniques à relever. Cela permet à l'équipe de travailler efficacement ensemble et de mieux communiquer, en évitant les malentendus et les erreurs.

Le travail d'équipe est essentiel pour atteindre des performances optimales dans tout domaine. Cependant, la collaboration entre individus peut être difficile et nécessite souvent des efforts supplémentaires pour assurer une communication claire et une coordination efficace.

Dans ce contexte, les équipes auto-gérées sont une méthode qui a fait ses preuves pour améliorer les performances et la satisfaction des membres de l'équipe. Elles sont devenues populaires en raison de leur capacité à encourager l'engagement des membres de l'équipe, à améliorer la qualité du travail et à favoriser l'innovation.

Cependant, même avec une équipe auto-gérée, la programmation peut souvent être un travail individuel qui peut être difficile à accomplir en équipe. Pour surmonter ce défi, la programmation par binôme est une méthode efficace pour travailler ensemble.

Dans les sections suivantes, nous allons explorer plus en détail les avantages et les inconvénients des équipes auto-gérées, ainsi que les principes de base de la programmation par binôme et les raisons pour lesquelles cette méthode peut améliorer la productivité et la qualité du code.

3.3.1 Performance et travail d'équipe

Cette section traite de la relation entre la performance et le travail d'équipe dans le contexte du développement logiciel agile. Les études montrent que les modèles mentaux partagés améliorent la qualité globale des logiciels en facilitant le partage d'informations, en réglant les conflits et en améliorant la compréhension commune des tâches et des objectifs. Les pratiques

agiles, telles que la métaphore du système et les stand-up journaliers, sont utiles pour développer des modèles mentaux partagés au sein d'une équipe. La présence d'un représentant du client au sein de l'équipe est également importante pour améliorer la performance de l'équipe.

Une définition commune d'une équipe est "un petit nombre de personnes aux compétences complémentaires qui s'engagent dans un but commun, un ensemble d'objectifs de performance et une approche pour lesquels ils se tiennent mutuellement responsables"[49].

Plusieurs études ont examiné si la théorie des modèles mentaux partagés était liée ou non à l'amélioration de la performance des équipes et à l'amélioration de modèles mentaux partagé[53, 42, 54]. Ces études ont constaté que les modèles mentaux partagés au sein des équipes de développement facilitent le partage d'informations entre les développeurs, règlent les conflits entre les représentants du client et les chefs de projet, de même qu'ils améliorent la qualité globale des logiciels. Rai et al[55] ont constaté que la présence d'un représentant du client au sein de l'équipe, plutôt que l'inverse, améliorait grandement les performances de celle-ci. En effet, en intégrant quelqu'un du métier un sentiment de confiance plus important était partagé parmi les membres, ce qui s'est avéré utile pour résoudre les différents conflits apparus pendant le développement.

Un autre exemple démontré dans l'étude de Zhang et al[56] est que les développeurs et les testeurs possèdent des objectifs différents. En effet, les développeurs souhaitent produire leur code aussi rapidement que possible tandis que les testeurs ont pour objectif de rendre un produit de haute qualité. Si ces deux profils avaient développé des modèles mentaux partagés ils auraient pu avoir une meilleure compréhension des objectifs de chaque groupe et éventuellement résoudre d'autres problèmes au sein du projet. Ces types de malentendus, qui peuvent avoir un impact négatif sur le résultat d'un processus de développement logiciel, sont en mesure d'être gérés en utilisant les pratiques agiles pour développer des modèles mentaux partagés au sein d'une équipe.

La théorie veut que les modèles mentaux partagés permettent aux équipes de s'adapter aux conditions changeantes des tâches lorsque la communication entre les membres de l'équipe est difficile[33]. Des études ont démontré que toutes les équipes ne développent pas une compréhension commune du travail en groupe et des tâches à accomplir au fil du temps[40]. Des interventions, ou pratiques, sont nécessaires pour formaliser la création d'une compréhension commune afin de créer une base de connaissances interne à l'équipe. Grâce au développement de modèles mentaux partagés, elles forment des attentes compatibles concernant la tâche et le groupe. Ces modèles mentaux partagés l'aident dans sa compréhension des événements présents et futurs ainsi que sur la raison de pourquoi ces évènements se sont produits. Par conséquent, pour agir de manière appropriée lorsque les clients changent les exigences, l'équipe agile doit développer un modèle mental partagé, puisque les pratiques agiles ont la capacité de l'aider à atteindre ce nouvel objectif.

On retrouve la théorie de Espinosa et al[42] sur la distinction entre les modèles mentaux de la tâche et le modèle mental du travail d'équipe. Les pratiques agiles facilitent la communication et la coordination des équipes afin de construire les objectifs des tâches partagées et de déterminer la manière de les atteindre. La théorie des modèles mentaux partagés insiste également sur l'importance de construire un modèle au sein de l'équipe. En effet l'utilisation efficace des pratiques agiles améliorera la qualité de la communication et de la coordination de l'équipe afin de s'assurer que ses membres ont une compréhension claire des rôles et des compétences de chacun.

Enfin l'étude de Yu et Petter [57] se concentre principalement sur trois aspect précis des méthodes agiles, à savoir la métaphore du système, les stand-up journaliers et la présence du client au sein de l'équipe.

— *La métaphore du système* est une pratique de la méthode agile eXtreme Programming. L'idée de la métaphore du système dans l'eXtreme Programming est assez simple : selon cette pratique, chaque morceau de code reçoit son propre nom. Ces noms doivent être compréhensibles pour toutes les parties prenantes du projet, y compris les développeurs, le client et les utilisateurs finaux. C'est pourquoi les programmeurs XP utilisent des métaphores pour les appeler. Les noms de ces parties de code ne peuvent pas être complexes. Ils ne doivent pas contenir de termes ou d'expressions techniques durs qui ne sont utilisés que par les développeurs de logiciels, car dans d'autres cas, le client pourrait ne pas les comprendre. Comme nous l'avons déjà mentionné, le client fait partie de l'équipe Extreme Programming. Il participe à toutes les étapes du processus de développement logiciel. C'est pourquoi les noms des éléments du produit doivent être simples et compréhensibles pour lui[6, 58]. Du point de vue de la théorie des modèles mentaux partagés, lorsqu'une équipe de développement agile utilise la pratique de la métaphore du système, elle développe un modèle mental partagé en employant naturellement la pratique de la planification des modèles mentaux partagés. La planification est une intervention utilisée dans les premières étapes du développement de l'équipe pour améliorer la compréhension partagée de la tâche et de l'équipe[59]. La pratique de la métaphore du système va de pair avec la pratique de la planification des modèles mentaux partagés. La pratique de la métaphore du système encourage les équipes agiles à créer un environnement compréhensible par tous et à utiliser des métaphores pour développer des compréhensions partagées concernant les objectifs du projet, les concepts clés, les fonctionnalités majeures, et les rôles et l'expertise des membres de l'équipe agile.

— *Les stand-up journaliers* est une des pratiques fondamentale de la méthode agile Scrum. Elle consiste en des réunions journalières de maximum quinze minutes et permettant aux membres de l'équipe de s'aligner sur le travail déjà effectué, ce qu'il reste à faire et les éventuels blocages. Les stand-up contribuent aux modèles mentaux partagés, la pratique agile de la réunion stand-up fournit une occasion d'augmenter les compréhensions partagées des équipes sur le travail de tâche par le suivi et le contrôle quotidiens de la progression du projet[60].

— *La présence du client* a une forte valeur ajoutée au sein de l'équipe. Cette pratique exige que le client soit disponible à temps plein sur site et puisse interagir à tout moment avec les développeurs. Son rôle est principalement de participer aux réunions de planification, aux futurs backlogs ou encore aux tests d'acceptance. En outre, le client sur site peut participer à la réunion quotidienne de stand-up. Le rôle du client est de spécifier la fonctionnalité, de hiérarchiser les exigences, d'effectuer les tests d'acceptation pour chaque version et de prendre les décisions commerciales finales[61]. La pratique agile du client sur site offre aux développeurs une plus grande opportunité d'apprendre ses besoins. En se référant aux quatre étapes du développement des modèles mentaux partagés (connaître, apprendre, comprendre et exécuter), la pratique agile du client sur site améliore les étapes de compréhension et d'exécution des équipes agiles. Grâce à la qualité et à la fréquence des échanges avec les clients, les développeurs ont plus d'occasions d'identifier si la fonctionnalité du système répond à leurs besoins, mais aussi de poser des questions sur les tâches en cours de développement. Ainsi, les développeurs de l'équipe agile acquièrent une compréhension partagée et précise de la tâche, ce qui lui permet d'exécuter le travail plus efficacement. La mise en œuvre de la pratique client sur site aide les développeurs des équipes agiles à construire des modèles mentaux de travail similaires et précis.

Les méthodes de développement logiciel agiles sont critiquées pour leur manque de fondement théorique. Cependant, la recherche de Yu et Petter [57] démontre comment appliquer une

théorie, les modèles mentaux partagés, pour comprendre la valeur des pratiques agiles dans le développement de niveaux plus élevés de collaboration dans un effort de développement logiciel.

3.3.2 Équipe auto-gérée

Dans la section précédente, nous avons abordé les principes de base des équipes agiles. Dans cette section, nous nous concentrerons sur un aspect crucial de l'équipe agile, à savoir la capacité de l'équipe à s'auto-gérer. Nous verrons que la transition vers une équipe auto-gérée peut-être l'un des plus grands défis lors de la mise en place d'une équipe agile. En effet, pour travailler efficacement comme une équipe agile, il est essentiel que les membres de l'équipe partagent des modèles mentaux communs. Nous explorerons également les facteurs qui peuvent empêcher les équipes de développer des modèles mentaux partagés et de s'auto-gérer efficacement.

Selon Moe et al[62] la transition vers des équipes auto-gérées constitue l'un des plus grands défis lors de la mise en place d'une équipe agile. En outre, il ne suffit pas de rassembler des individus dans un groupe, de les étiqueter " agile " et de les faire travailler ensemble pour espérer les voir automatiquement se coordonner et travailler efficacement comme une équipe agile.

Dans une équipe logicielle, les membres sont conjointement responsables du produit final et doivent développer des modèles mentaux partagés en négociant des compréhensions communes à la fois sur le travail d'équipe et la tâche[40]. En effet, les objectifs du projets, les exigences du système et des clients, les responsabilités et rôles des membres de l'équipe doivent être compris par toutes les parties concernées [63].

Travailler en équipe exige d'avoir des modèles mentaux commun[24]. Les travaux de Salas et al[24] explique que si cette compréhension des modèles mentaux communs n'est pas atteinte par l'ensemble de l'équipe, les membres individuels peuvent se diriger vers des objectifs différents, ce qui à son tour, conduira à un retour d'information ou à une assistance inefficace ou inexistante.

Les modèles mentaux partagés sont une condition préalable à la communication, au suivi et à l'organisation de l'équipe. Les résultats des études de Moe et al[62] et de Levesque et al[40] montrent les mêmes résultats, à savoir que toutes les équipes ne développent pas des modèles mentaux de plus en plus partagés au fil du temps. Ceci peut être expliqué par un blocage au sein de l'entreprise, celle-ci ayant des difficultés à se détacher des pratiques de gestion de projet plus traditionnelles au lieu d'ajuster le processus de développement en faisant de la place pour la réflexion et l'apprentissage.

3.3.3 Programmation par binôme

Souvent cité comme pilier des méthodes agiles, plus particulièrement l'eXtreme Programming, la programmation par binôme est au centre de l'étude de Kude et al[64]. Les études précédentes s'accordent en général pour dire que la programmation par binôme peut augmenter la qualité du code logiciel développé et la rapidité avec laquelle la tâche est exécutée. Néanmoins, ces études mettent en avant les performances individuelles et dyadiques des développeurs[65] et non l'équipe dans son ensemble et l'influence de cette méthode sur la perception des modèles mentaux partagés par ses membres. Par exemple, les travaux antérieurs ont étudié la qualité du code développé par les paires par rapport aux individus, la rapidité respective avec laquelle le code est achevé, et l'effort de la programmation en binôme par rapport à la programmation en solo[66].

La figure ci-dessus montre le modèle de recherche proposé par Kude et al[64]. Le principal argument est que la programmation en binôme crée des effets sur la performance au niveau

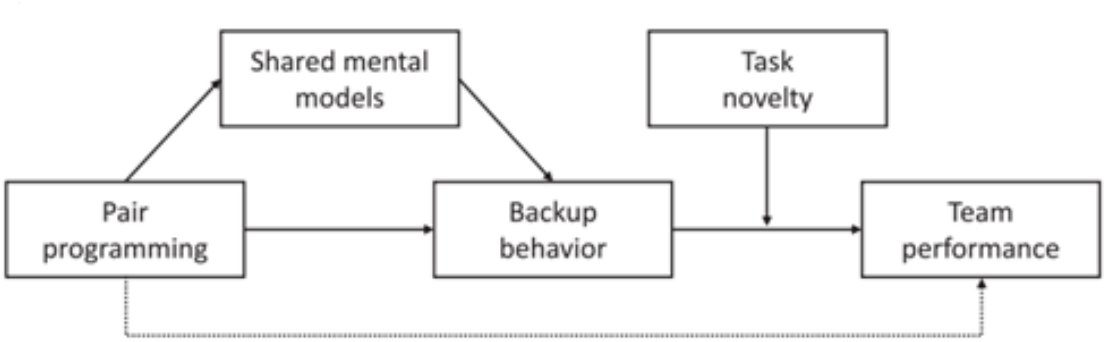

FIGURE 4 – Modèle de recherche [64]

de l'équipe en encourageant le comportement de sauvegarde parmi ses membres. La programmation en binôme augmente les modèles mentaux partagés entre les membres de l'équipe et, par conséquent, le comportement de sauvegarde dans les équipes. En retour, le comportement de sauvegarde aide les équipes à atténuer l'effet négatif de la nouveauté de la tâche sur la performance.

Mais comment la programmation par binôme influence-t-elle les modèles mentaux de l'équipe ? Les pratiques de développement de logiciels contemporains sont principalement axées sur l'interaction et la communication pendant les phases de planification et d'évaluation. Par exemple la méthodologie Scrum préconise d'organiser des réunions journalières de quelques minutes afin d'aligner les membres de l'équipe sur les tâches en cours et à venir. Néanmoins, ces interactions sont moins présentes voir inexistante dans les phases de codage, principalement réalisées par un individu seul. En revanche, lorsque l'on s'appuie sur la programmation en binôme, les développeurs de logiciels s'engagent dans une observation et des discussions pendant les phases de codage, ce qui devrait renforcer les modèles mentaux partagés des membres de l'équipe. Par exemple, la programmation en binôme contribue probablement à la convergence des visions des développeurs impliqués concernant le développement futur du logiciel ainsi que son architecture sous-jacente. De plus il est démontré que ces modèles se propagent au niveau de l'équipe quand un membre du binôme vient à changer ou que le code produit se retrouve partagé, commenté ou encore documenté[67].

3.4 Domain Driven Design

Dans cette étude, nous explorons la relation entre le DDD et les modèles mentaux partagés au sein des équipes de développement logiciel. Bien que les concepts clés de DDD et les modèles mentaux partagés aient été présentés précédemment dans ce travail, il est important d'examiner comment ces deux éléments s'entrecroisent et se renforcent mutuellement pour améliorer la collaboration, la communication et l'efficacité des équipes de développement.

Les modèles mentaux partagés jouent un rôle crucial dans la compréhension mutuelle des problèmes et des solutions au sein d'une équipe. Ils facilitent la coordination et la coopération, réduisent les ambiguïtés et les malentendus, et permettent une meilleure prise de décision. De même, DDD vise à aligner étroitement le développement logiciel sur les besoins métier en mettant l'accent sur la modélisation du domaine et la communication efficace entre les membres de l'équipe.

Cependant, les liens entre les modèles mentaux partagés et DDD ne sont pas directement

établis dans la littérature existante. Dans cet état de l'art, nous examinons comment les concepts et les pratiques de DDD contribuent à la création et au maintien de modèles mentaux partagés. Nous nous appuierons sur les travaux d'auteurs tels que Widjaja et al., Hendrickson et Bate, Millett et Tune, Haywood, Nilsson, et d'autres pour explorer cette relation.

Nous commencerons par analyser les concepts et pratiques DDD qui favorisent les modèles mentaux partagés, tels que l'Ubiquitous Language et les Bounded Contexts. Ensuite, nous examinerons des études de cas et des expériences pratiques pour illustrer comment DDD et les modèles mentaux partagés se manifestent dans des projets réels. Enfin, nous discuterons des méthodologies, des outils et des techniques qui soutiennent la création et le maintien de modèles mentaux partagés dans le contexte du DDD, ainsi que des perspectives futures et des directions de recherche pour approfondir notre compréhension de cette relation.

3.4.1 Concepts et pratiques DDD favorisant les modèles mentaux partagés

Dans cette section, nous allons aborder le points de vue de plusieurs auteurs sur les artefacts majeurs qui caractérisent le DDD. Nous commencerons par discuter de l'Ubiquitous langage et de la facilité qu'il apporte aux équipes. Nous aborderons ensuite l'intérêt de séparer les différents contextes métier de l'application en Bounded Context et nous terminerons par discuter de l'importance de la communication et de la collaboration dans les projets DDD.

Ubiquitous Langage

L'Ubiquitous Language est un concept clé du DDD qui favorise la création de modèles mentaux partagés au sein des équipes de développement logiciel. Il s'agit d'un langage commun élaboré par les membres de l'équipe, y compris les développeurs, les experts métier et les autres parties prenantes, pour décrire les concepts du domaine et faciliter la communication[31].

L'importance de l'Ubiquitous Language dans la promotion des modèles mentaux partagés est soulignée par plusieurs auteurs. Par exemple, Millett et Tune[68] expliquent que l'Ubiquitous Language permet de réduire les ambiguïtés et les malentendus en assurant que chaque membre de l'équipe utilise les mêmes termes et concepts pour décrire le domaine. Dans leur ouvrage "Patterns, Principles, and Practices of Domain-Driven Design", Millett et Tune expliquent que l'Ubiquitous Language est un langage commun utilisé par tous les membres d'une équipe pour décrire le domaine métier. Cette approche permet de réduire les ambiguïtés et les malentendus en assurant que chaque membre de l'équipe utilise les mêmes termes et concepts pour décrire le domaine.

L'Ubiquitous Language est conçu pour faciliter la communication entre les membres de l'équipe, en particulier entre les experts métier et les développeurs. Il aide à établir une compréhension commune du domaine et à identifier les problèmes de communication qui peuvent survenir en raison de l'utilisation de termes ou de concepts différents.

Millett et Tune soulignent que le langage doit être constamment affiné et mis à jour pour refléter l'évolution du domaine et des connaissances de l'équipe. L'Ubiquitous Language doit être utilisé dans toutes les conversations, les documents et le code pour garantir une compréhension cohérente et partagée.

En créant et en maintenant un langage commun, l'équipe peut travailler de manière plus efficace et éviter les erreurs et les confusions qui peuvent survenir lorsqu'il y a un manque de compréhension partagée. L'Ubiquitous Language favorise ainsi la collaboration et l'alignement entre les membres de l'équipe, ce qui est essentiel pour réussir dans le développement de logiciels orientés domaine.

Widjaja et al[69] soutiennent également que l'Ubiquitous Language est essentiel pour créer une cohésion au sein de l'équipe. Ils mettent en évidence l'importance de la collaboration étroite entre les experts métier et les développeurs pour élaborer ce langage commun, ce qui permet de garantir que le modèle de domaine reflète fidèlement les besoins et les exigences des utilisateurs.

De même, Haywood[70] souligne que l'Ubiquitous Language joue un rôle crucial dans la facilitation de la communication. Il insiste sur le fait que le langage doit être cohérent et compris par tous les membres de l'équipe, afin de garantir une collaboration efficace et une prise de décision éclairée.

Bounded Context

Les Bounded Contexts jouent un rôle crucial dans la promotion de la communication au sein des équipes travaillant avec le DDD. Un Bounded Context est une limite conceptuelle qui permet de découper un domaine complexe en sous-domaines plus gérables, favorisant ainsi une meilleure compréhension de chaque sous-domaine par les membres de l'équipe[31].

Parmi les auteurs abordant ce concept, Nilsson[71] souligne l'importance des Bounded Contexts pour éviter les conflits et les malentendus qui pourraient survenir lorsque des termes et des concepts sont utilisés de manière ambiguë dans des contextes différents. En délimitant clairement les contextes, les équipes peuvent travailler de manière plus cohérente et harmonieuse, en ayant une meilleure compréhension des responsabilités et des interactions entre les différents contextes. Nilsson suggère que l'utilisation appropriée des Bounded Contexts peut améliorer la communication entre les membres de l'équipe et réduire les chances de malentendus en clarifiant les frontières des sous-domaines et en assurant une meilleure cohésion dans la compréhension du domaine métier.

Selon Millett et Tune [72], les Bounded Contexts aident à préserver la cohérence des modèles de domaine en isolant les modèles dans des contextes délimités. Les contextes délimités sont utilisés pour distinguer clairement les différentes parties du système, chacune ayant sa propre logique et ses propres règles métier. Cela permet aux membres de l'équipe de se concentrer sur un sous-domaine spécifique et de développer une compréhension partagée de ses règles et de ses interactions avec d'autres contextes délimités.

Widjaja et al[69] discutent de l'importance des Bounded Contexts dans le cadre du DDD et de leur rôle pour assurer une meilleure compréhension et une gestion de la complexité du domaine. Ils soutiennent que les Bounded Contexts contribuent à clarifier les responsabilités, les interactions et les modèles de domaine spécifiques pour chaque sous-domaine. Cela permet aux membres de l'équipe de mieux comprendre et gérer les aspects complexes du domaine, favorisant ainsi une communication et une collaboration plus efficaces.

Enfin Haywood [70] aborde l'importance des Bounded Contexts dans le DDD. Il souligne que les Bounded Contexts aident à découper un domaine complexe en sous-domaines plus gérables, en isolant les modèles de domaine et en préservant leur cohérence. Selon Haywood, les Bounded Contexts jouent un rôle essentiel pour éviter les conflits et les malentendus qui peuvent survenir lorsque des termes et des concepts sont utilisés de manière ambiguë dans différents contextes. En délimitant clairement les contextes, les équipes peuvent travailler de manière plus cohérente et harmonieuse, en ayant une meilleure compréhension des responsabilités et des interactions entre les différents contextes délimités.

Collaboration et communication

Dans le contexte du DDD, plusieurs auteurs ont abordé l'importance de la collaboration et de la communication pour favoriser à compréhension mutuelle des concepts du domaine.

Hendrickson et Bate [73] mettent l'accent sur l'importance de la collaboration et de la communication dans les projets agiles et DDD. Ils soutiennent que ces approches visent à améliorer la collaboration entre les membres de l'équipe en les encourageant à travailler ensemble étroitement pour comprendre et résoudre les problèmes du domaine. Ils soulignent également l'importance de la communication claire et efficace, en particulier en ce qui concerne le partage de la compréhension du domaine entre les membres de l'équipe et les parties prenantes.

Millett et Tune [72] insistent sur la nécessité d'une communication claire et efficace pour permettre aux membres de l'équipe de partager leurs connaissances et de collaborer sur le développement de solutions adaptées aux besoins du domaine. Ils considèrent que la communication et la collaboration sont essentielles pour favoriser la compréhension mutuelle des concepts du domaine et pour résoudre les problèmes de manière cohérente.

Enfin Remy et Zayni [74] abordent également l'importance de la collaboration et de la communication dans le DDD. Selon eux, les équipes doivent travailler ensemble pour développer des solutions adaptées aux problèmes du domaine, en s'appuyant sur la compréhension mutuelle des concepts et des modèles mentaux partagés. Ils soutiennent que la communication est un élément essentiel pour favoriser la collaboration et pour s'assurer que les membres de l'équipe comprennent et respectent les modèles de domaine définis dans le cadre du DDD.

3.4.2 Méthodologies et outils

La mise en place de modèles mentaux partagés dans le cadre du DDD est soutenue par diverses méthodologies et outils. Parmi ces derniers, les techniques de modélisation collaborative jouent un rôle clé dans la création d'une compréhension partagée entre les membres de l'équipe et les parties prenantes. Hendrickson et Bate[73] soulignent l'importance de la collaboration et des ateliers de modélisation pour favoriser la compréhension mutuelle et la création de solutions adaptées aux besoins du domaine.

Dans leur ouvrage "Agile Retrospectives : Making Good Teams Great" [73], Hendrickson et Bate mettent l'accent sur l'importance de la collaboration et des ateliers de modélisation dans le cadre des méthodologies agiles. Bien que leur travail se concentre principalement sur les rétrospectives agiles, les idées qu'ils présentent peuvent également s'appliquer au DDD.

Les auteurs soulignent que les ateliers de modélisation et la collaboration sont essentiels pour créer une compréhension partagée du domaine métier parmi les membres de l'équipe. Ils soutiennent que ces ateliers permettent à l'équipe de mieux comprendre les besoins du client et d'identifier les problèmes potentiels avant qu'ils ne deviennent des obstacles majeurs. De plus, la collaboration étroite entre les membres de l'équipe et les parties prenantes du projet peut aider à réduire les malentendus et à assurer une communication claire et efficace.

En organisant des ateliers de modélisation et en encourageant la collaboration, les équipes de développement peuvent créer des modèles mentaux partagés qui leur permettent de mieux comprendre et de résoudre les problèmes complexes liés à leur domaine. Cela, à son tour, peut mener à une conception et une implémentation de logiciels plus efficaces et de meilleure qualité. En fin de compte, l'accent mis par Hendrickson et Bate sur la collaboration et les ateliers de modélisation souligne l'importance de ces pratiques pour favoriser la compréhension globale de l'équipe et contribuer au succès global des projets de développement logiciel utilisant le DDD.

De même, Millett et Tune[75] insistent sur l'importance des ateliers de modélisation pour faciliter la collaboration et la communication entre les membres de l'équipe. Ils recommandent l'utilisation de techniques telles que l'Event Storming, une méthode de modélisation collaborative développée par Albert Brandolini[76], pour permettre aux membres de l'équipe de travailler

ensemble sur la modélisation du domaine et d'explorer les différents scénarios et événements qui se produisent dans le domaine.

Enfin, Remy et Zayni[74] abordent également l'importance des méthodologies collaboratives dans le cadre du DDD. Ils suggèrent également l'adoption de techniques de modélisation collaborative, telles que l'Event Storming, qui peut aider à créer une compréhension partagée du domaine et à réduire les ambiguïtés et les malentendus. En outre, ces techniques permettent aux membres de l'équipe d'identifier les problèmes potentiels et de trouver des solutions adaptées avant de commencer le développement.

3.5 Conclusion de l'état de l'art et justification de la recherche

Nos différentes lectures montrent qu'une compréhension poussée des modèles mentaux partagés influence favorablement les équipes de développement logiciel dans leur travail.

Nous notons tout particulièrement que les méthodologies Agiles représentent un facteur de réussite important dans un projet, tant celles-ci influencent la bonne évolution des modèles mentaux partagés de l'équipe. En effet, les stand-up quotidiens, la présence du client dans l'équipe de développement ou encore la programmation en binôme ne sont que quelques exemples de pratiques Agile bénéfiques à la normalisation et à l'uniformisation des modèles mentaux de l'équipe.

De manière générale, la communication et le travail d'équipe restent des aspects récurrents des cas d'études, Agiles ou non. En effet, contrairement aux idées reçues, les métiers entourant le développement logiciel sont profondément sociaux. Une communication fréquente entre les membres de ces équipes ne peut donc être que bénéfique pour le projet.

Nous constatons néanmoins un manque criant d'études consacrées à des méthodologies autres que celles Agiles utilisés par les développeurs afin d'aligner leurs connaissances et donc leurs modèles mentaux partagés. Plus précisément, nous avons constaté l'absence de liens directs entre les modèles mentaux partagées et l'approche DDD, même si certaines pistes énoncées plus haut laissent à penser qu'il existe une influence mutuelle.

L'absence de littérature sur les sujets précités peut sembler surprenante, mais se comprend assez facilement : les thèmes abordés dans cet état de l'art sont récents et en évolution constante. La manière d'approcher le développement logiciel change tout aussi rapidement que l'informatique elle-même. Il est donc difficile d'appréhender sur le long terme ces méthodologies en constante évolution, bien que certaines aient su faire l'unanimité et se maintenir au fil des années, comme c'est le cas pour les méthodologies Agiles.

Dès lors, l'application de l'approche DDD et son influence sur les modèles mentaux partagés de l'équipe peut faire l'objet du présent travail pour les raisons suivantes :

— Il est constatée une absence totale de recherche concernant l'influence du Domain Driven Design sur les modèles mentaux ;

— Il existe une méconnaissance des acteurs des secteurs de l'IT concernant le Domain Driven Design qui mériterait à notre sens d'être mieux mise en valeur ;

4 Problématique et questions de recherche

4.1 Problématique

Le processus de développement logiciel est complexe et nécessite la collaboration de nombreux acteurs, tels que les développeurs, les analystes, les experts métier, les architectes, les testeurs et les utilisateurs finaux. Chaque acteur possède sa propre perspective et compréhension des objectifs, des exigences et des contraintes du système, ainsi que des rôles et responsabilités de chacun.

Malgré l'adoption croissante des méthodologies Agiles dans les projets de développement logiciel, les équipes continuent de faire face à des défis importants en termes de communication, de compréhension commune des objectifs et de coordination efficace. Ces défis peuvent entraver la performance de l'équipe et retarder la livraison du projet. Les modèles mentaux partagés pourraient ici jouer un rôle essentiel.

Parce que l'approche DDD se concentre sur la modélisation du domaine métier du système d'information en utilisant un langage métier commun pour faciliter la communication et la compréhension entre les différents acteurs, cette recherche se propose d'étudier l'impact de cette approche sur l'émergence et l'évolution des modèles mentaux partagés au sein des équipes de développement. Deux points de vue sont considérés : (1) l'impact de l'approche DDD sur les modèles mentaux partagés ; (2) l'impact de la combinaison approche DDD-méthodologies Agiles sur ces modèles.

4.2 Questions de recherche

La première question de recherche de cette étude est :

Quelle est l'influence de l'approche Domain Driven Design sur l'évolution des modèles mentaux partagés entre les différents acteurs impliqués dans les phases de conception et la mise en œuvre d'un développement logiciel ?

La seconde question de recherche, qui découle de la première, est :

Comment l'application conjointe de l'approche Domain Driven Design et des méthodologies Agiles influence-t-elle l'évolution des modèles mentaux partagés entre les différents acteurs impliqués dans la conception et la mise en œuvre d'un développement logiciel ?

Pour répondre à ces questions, il convient d'examiner les processus de collaboration, de communication et de coordination entre les différents acteurs, ainsi que les impacts de l'approche DDD et de sa combinaison avec les méthodologies Agiles sur la qualité et la pertinence des modèles mentaux partagés.

5 Méthode

Ce chapitre s'arrête sur la méthodologie utilisée pour l'analyse du cas pratique, à savoir la société-cible. Cette analyse va permettre de confirmer ou d'infirmer les intérêts et les enjeux, présentés dans le chapitre précédent, d'adopter l'approche DDD conjointement aux méthodes agiles afin de faire évoluer les modèles mentaux partagés de l'équipe. Le contexte du cas pratique est brièvement présenté ainsi que l'échantillon sur lequel se base l'étude. En outre, le développement du guide d'entretiens, utile à la collecte de données qualitatives, est détaillé.

5.1 Contexte de l'étude

L'étude de terrain a été menée au sein de l'entreprise I-Pulses, créée en 2018 et spécialisée dans le Data Management et la Business Intelligence. Elle compte actuellement une vingtaine d'employés qui occupent des postes de chefs de projet, d'analystes ou encore de développeurs. Elle a introduit l'approche DDD afin d'améliorer sa capacité à livrer un produit à forte complexité métier dans les délais, d'accroître la qualité et d'améliorer le travail d'équipe sur le dernier projet en date. Ce projet, "Gerico", est donc le premier à bénéficier de la combinaison approche DDD-méthode Agile.

L'objectif de "Gerico" est de développer un système de gestion de clients et de dossiers pour une étude d'huissiers. Cette application doit prendre en compte la gestion des différents intervenants au sein d'un dossier - tels le débiteur, le requérant, l'administrateur de biens ou encore l'avocat – la gestion des dossiers et le suivi de ceux-ci, la gestion de la comptabilité de l'étude, etc.

5.2 Récolte des données qualitatives

Compte tenu de la situation dans l'entreprise-cible et de la diversité des profils accessibles, une approche qualitative de récolte de données a été envisagée. Cette approche semblait la plus adaptée pour comprendre le ressenti des différentes personnes interrogées et évaluer les pratiques de l'entreprise[77]. Trois étapes ont été nécessaires pour collecter les données : (1) la constitution de l'échantillon des personnes-cible, (2) la préparation d'un guide d'entretiens et (3) la réalisation des entretiens.

5.2.1 Constitution de l'échantillon

Dans le but d'obtenir une perspective diversifiée, nous avons mené des entretiens avec cinq développeurs et analystes, en veillant à inclure un éventail d'âges (de 24 à 50 ans), de niveaux d'expérience (de 2 à 25 ans), de rôles (analyste, développeur, product owner, architecte) et d'expérience spécifique avec l'approche DDD. Pour chaque entretien, nous avons utilisé un guide semi-structuré, puis transcrit intégralement les échanges pour permettre une analyse approfondie.

Le nombre d'entretiens utiles à la recherche a été fixé par saturation empirique[78]. Ainsi, au terme des cinq entretiens, les réponses collectées auprès des intervenants se répétaient déjà.

Néanmoins, certaines nuances sont perceptibles en fonction du profil de la personne interviewée. Nous avons décidé de répertorier les différents profils rencontrés ci-dessous en citant leurs caractéristiques afin de pouvoir plus facilement les référencer dans les résultats et la discussion.

— profil 1 : lead analyste senior de 40 ans (Annexe II8.2) ;

— profil 2 : développeur back-end medior de 30 ans (Annexe III8.3) ;
— profil 3 : développeur back-end junior de 24 ans (Annexe IV8.4) ;
— profil 4 : analyste data science junior de 27 ans (Annexe V8.5) ;
— profil 5 : lead développeur back-end senior de 48 ans (Annexe VI8.6)

5.2.2 Mise au point du guide d'entretiens

L'entretien semi-directif est une technique de collecte de données qui contribue au développement de connaissances favorisant des approches qualitatives et interprétatives relevant en particulier des paradigmes constructivistes[79]. Si plusieurs possibilités s'offraient à nous pour recueillir les informations nécessaires à notre étude, les entretiens semblaient la meilleure option compte tenu du petit nombre de travailleurs dans l'entreprise-cible[80]. L'objectif d'un entretien est ici d'analyser les perceptions et les significations que les personnes donnent aux évènements et aux pratiques de l'entreprise[81].

Afin de préparer les entretiens, un guide de type semi-directif a été développé. Ce guide se compose principalement de trois grands axes :

— Une introduction et une mise en contexte permettant d'expliquer les objectifs de l'entretien ;
— Une première série de questions commune à tous les profils, reprenant le ressenti général des répondants sur l'utilisation de l'approche DDD ;
— Deux séries de questions, l'une orientée analyse et l'autre développement, afin de mieux cibler l'influence de l'approche en fonction des responsabilités du répondant.

Ce guide a permis d'aborder les mêmes sujets généraux avec tous les répondants, mais aussi d'approfondir certaines thématiques selon le poste occupé par l'interviewé. En effet, en fonction de leur position, les répondants ont parlé davantage de l'un ou l'autre aspect de leur vie professionnelle.

Nous vous présentons, ci-dessous, les différentes questions composant le guide d'entretiens.

Q1	Warm-up	Sur quoi travaillez-vous actuellement ?
Q2	Warm-up	Quel est l'état d'avancement du projet ?

⟶ Les questions *Warm-up* servent d'introduction à l'entretien et nous permettent également de donner le contexte de notre étude.

Q3	Généralité	Comment définiriez-vous la complexité du domaine métier du projet ?
Q4	Généralité	Avez-vous déjà travaillé dans des projets d'une complexité égale ou supérieur ?
Q5	Généralité	Comment décririez-vous votre niveau de compréhension de celui-ci ?
Q6	Généralité	Comment était-il sur vos précédents projets ?
Q7	Généralité	Que pensez-vous du niveau de compréhension de vos collègues ?
Q8	Généralité	Pensez-vous avoir une compréhension commune des termes métiers ?
Q9	Généralité	Était-ce le cas dans les projets antérieurs ?

⟶ Nous désirons ici mettre en avant la complexité métier de l'application-cible ainsi que la compréhension qu'en ont les intervenants. En effet, l'approche DDD a pour but de casser la complexité métier comme le précise son auteur Eric Evans[31].

Q10	Généralité	Comment découvrez-vous les changements dans le projet ?
Q11	Généralité	Comment cela a-t-il été fait dans les projets antérieurs ?
Q12	Généralité	Comment faites-vous face aux changements dans le projet ?
Q13	Généralité	Comment cela a-t-il été fait dans des projets antérieurs ?

⟶ Ces questions proviennent de l'article de Moe, Dingsøyr et Dybå[62] et traitent du changement et de la cohésion d'équipe en méthode Agile. Ces questions sont-elles mêmes inspirées du travail de Dickinson and McIntyre[82] sur les modèles d'équipe de travail. Nous avons décidé de les intégrer afin de comprendre comment l'approche DDD pouvait influencer un projet utilisant les méthodes agiles.

Q14	Généralité	L'établissement d'un Ubiquitous Language a-t-il facilité votre compréhension du métier et des objectifs attendus ?
Q15	Généralité	Comment cela a-t-il été fait dans des projets antérieurs ?
Q16	Généralité	L'utilisation du Tactical Pattern vous aide-t-il dans la compréhension du métier ?
Q17	Généralité	Comment cela a-t-il été fait dans les projets antérieurs ?
Q18	Généralité	L'utilisation de Context Map et des Bounded Context vous aide-il dans la compréhension du métier ?
Q19	Généralité	Comment cela a-t-il été fait dans des projets antérieurs ?

⟶ L'Ubiquitous Langage, le Tactical Pattern, les Context Map et les Bounded Context sont les artefacts principaux de l'approche DDD que nous avons décidé de mettre en avant. Grâce à ces questions, nous souhaitons directement mettre en relation ces différents artefacts à la compréhension du domaine métier de l'équipe.

Q21	Généralité	L'équipe a-t-elle un objectif commun pour le projet ?
Q22	Généralité	Les projets précédents avaient-ils un objectif commun ?
Q23	Généralité	Tout le monde connaît-il le résultat attendu du projet ?
Q24	Généralité	Comment est la performance de l'équipe ?
Q25	Généralité	Comment cela a-t-il été fait dans les projets précédents ?
Q26	Généralité	Qu'est-ce qui fonctionne ?
Q27	Généralité	Qu'est-ce qui ne fonctionne pas ?

⟶ Ces questions proviennent de l'article de Moe, Dingsøyr et Dybå[62] et traitent de la performance et des objectifs du projet. Nous souhaitons ici démontrer l'impact de l'approche DDD sur la vélocité de l'équipe et le ressenti des membres sur la cohérence du projet.

Q28	Analyste	Combien de temps vous a-t-il été laissé pour pré analyser le métier avant le lancement du projet ?
Q29	Analyste	Comment cela a été fait dans les projets antérieurs ?
Q30	Analyste	Communiquez-vous régulièrement avec un expert métier ?
Q31	Analyste	Comment cela a été fait dans les projets antérieurs ?
Q32	Analyste	L'Event storming vous a-t-il aidé dans la construction de l'Ubiquitous langage ?
Q33	Analyste	Comment cela a été fait dans les projets antérieurs ?
Q34	Analyste	Déterminer les VO, Entities, etc... pour un analyste est-il pertinent ? En quoi ? Plus-value ?
Q35	Analyste	Comment cela a été fait dans les projets antérieurs ?
Q36	Analyste	Vous arrive-t-il d'effectuer des corrections dans les analyses réalisées ? A quelle fréquence ?
Q37	Analyste	Comment cela a été fait dans les projets antérieurs ?

⟶ Les questions ci-dessus sont spécifiquement adressées aux analystes de l'équipe et ciblent les aspects des artefacts de l'approche DDD ayant le plus d'impact sur le domaine métier.

Q38	Developpeur	Comment les informations métiers vous sont-elles transmises ?
Q39	Developpeur	Comment cela a été fait dans les projets antérieurs ?
Q40	Developpeur	Jugez-vous la découpe en VO, Entities, etc pertinente pour le développement ?
Q41	Developpeur	Comment cela a été fait dans les projets antérieurs ?
Q42	Developpeur	Comment jugez-vous la lisibilité du code au travers des découpes précitées ?
Q43	Developpeur	Comment cela a été fait dans les projets antérieurs ?
Q44	Developpeur	Pouvez-vous facilement lire, comprendre et/ou reprendre le code d'un de vos collègues ?
Q45	Developpeur	Comment cela a été fait dans les projets antérieurs ?

$\longrightarrow$ Enfin, cette partie du questionnaire est spécifiquement adressée aux développeurs de l'équipe et cible les aspects plus techniques de l'approche DDD et l'influence de ceux-ci sur leur compréhension du domaine métier.

Ainsi construit, le guide d'entretien couvre les composantes du modèle de Dickinson et McIntyre, les questions relatives à la méthode agile Scrum [82] ainsi que nos propres développements suite à la lecture du livre de Evans [31].

5.2.3 Réalisation des entretiens

Les entretiens ont eu lieu dans les locaux d'I-Pulses et ont duré entre trente et soixante minutes. Les retranscriptions sont en annexes 8.

5.3 Traitement et analyse des données

Les données qualitatives récoltées ont été analysées selon un codage ouvert, suivi d'un codage axial. Ces étapes consistent à coder les données, une pratique qui comporte l'identification et l'étiquetage des thèmes récurrents et significatifs. Ce processus de codage s'est déroulé en deux phases : une phase de codage ouvert pour balayer largement les données, suivie d'une phase de codage axial pour relier les thèmes et identifier les relations entre eux8.7.

Nous avons ensuite classé le résultat du codage axial en plusieurs thématiques :

— Établissement d'un Ubiquitous Langage pour une compréhension commune ;

— Classification en Bounded Context/Context Map ;

— Plus-value de l'utilisation du Tactical Patterns dans la compréhension générale ;

— La complexité du DDD en vue d'une simplification métier ;

— Approche conjointe DDD-Agile positive ;

Ces différentes thématiques serviront de structure pour expliquer les résultats et pour organiser les discussions.

6 Résultats et discussions

Cette étude, centrée sur l'impact du DDD dans le domaine du développement logiciel, aborde comment le DDD peut influencer et favoriser l'évolution des modèles mentaux partagés au sein des équipes de développement. Notre objectif principal est de déterminer si le DDD, en tant qu'approche, peut améliorer la communication et la compréhension des concepts parmi les équipes engagées dans des projets complexes.

Les résultats obtenus suite aux cinq entretiens conduits soutiennent cette hypothèse. Ils mettent en évidence des aspects clés du DDD qui contribuent à cette amélioration. Ces résultats sont regroupés en cinq grandes thématiques, identifiées lors du codage des entretiens : (1) l'importance de l'Ubiquitous Language pour une communication fluide et efficace, (2) la pertinence de la classification en Bounded Context et Context Map pour clarifier les responsabilités et les interactions, (3) l'apport des Tactical Patterns pour une compréhension améliorée des concepts métier, (4) le caractère complexe, mais enrichissant du DDD, et (5) l'impact de l'agilité sur les projets DDD.

Chaque section de ce chapitre commence par une introduction faisant le lien entre les concepts théoriques explicités précédemment et les questions posées aux intervenants. Les réponses obtenues ont été ensuite catégorisées en fonction de leur orientation plus générale à l'équipe ou plus spécifique aux analystes et aux développeurs. Les discussions qui suivent représentent une progression naturelle de nos premières observations et visent à apporter des réponses à notre problématique initiale, notamment en explorant plus profondément les implications de ces observations pour l'utilisation du DDD dans les équipes de développement logiciel.

Cette analyse conjointe des résultats et de leur discussion permet de mettre en lumière les éléments clés du DDD qui favorisent une meilleure compréhension et une évolution des modèles mentaux partagés au sein des équipes de développement logiciel. Elle aborde également les défis et les complexités associés à l'adoption de cette approche, ainsi que de la manière dont l'agilité peut renforcer les bénéfices du DDD dans la gestion de projets logiciels.

6.1 L'Ubiquitous Language comme vecteur de communication

L'Ubiquitous Language, un concept central du DDD introduit par Eric Evans, désigne la création d'un langage commun et précis partagé par tous les membres de l'équipe de développement ainsi que les experts métiers impliqués dans le projet. Il s'appuie sur le modèle de domaine employé dans le logiciel, requérant une grande rigueur pour éviter les ambiguïtés mal tolérées par les logiciels.

Evans souligne que l'emploi de ce langage universel lors des échanges avec les experts du domaine est crucial pour tester, valider ce langage, et par extension, le modèle de domaine. Il insiste également sur la nécessité de faire évoluer ce langage (et le modèle) à mesure que l'équipe approfondit sa compréhension du domaine concerné.

"By using the model-based language pervasively and not being satisfied until it flows, we approach a model that is complete and comprehensible, made up of simple elements that combine to express complex ideas.

...

Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding ; developers should watch for ambiguity or inconsistency that will trip up design."

– Eric Evans[31]

Dans le cadre de notre étude, nous avons interrogé les cinq intervenants pour recueillir leur ressenti sur leur compréhension du métier et des objectifs à atteindre depuis l'introduction d'un Ubiquitous Language (Q14 et Q15). Nous nous attarderons également sur le processus de construction de cet Ubiquitous Language par les analystes lors des sessions d'Event Storming avec les experts métiers (Q32 et Q33).

Notre argumentation, précédée par une présentation des résultats généraux puis des points de vue spécifiques des analystes et des développeurs, se concentrera sur deux axes essentiels : l'influence de l'Ubiquitous Language sur la communication au sein de l'équipe et son impact sur la formation des modèles mentaux partagés.

6.1.1 Généralités

Au cours des entretiens menés, il est apparu clairement que l'un des éléments essentiels pour le bon fonctionnement de l'équipe était la création d'un langage commun partagé entre tous les membres, en particulier entre les analystes et les développeurs. Cette observation a été soulignée par l'intervenant 4, qui a mis en avant l'importance de cette cohérence linguistique pour faciliter la communication, la collaboration et la compréhension mutuelle entre les différents acteurs impliqués dans le projet :

"On voulait que tout le monde comprenne bien les implications métiers, les analystes bien sûr, mais également les développeurs. Donc oui, il y a bien un langage commun, en tout cas, on essaie un maximum".

En effet, la complexité des projets informatiques et la diversité des compétences et des connaissances des intervenants peuvent parfois rendre difficile la transmission des informations et la compréhension des différents enjeux [42]. Dans ce contexte, la mise en place d'un vocabulaire commun permet de clarifier les échanges et de faciliter la résolution des problèmes en évitant les malentendus et les incompréhensions.

Les participants ont également souligné que l'Ubiquitous Language permettait de clarifier des concepts métiers complexes et simplifiait les discussions entre collègues, comme le mentionne l'extrait de l'entretien 3 suivants :

"J'ai trouvé les interactions entre analystes, développeurs et experts métier beaucoup plus claires que ce que j'ai connu précédemment. On avait l'impression de parler de la même chose et donc les discussions avec nos collègues étaient beaucoup plus simples."

Ce dernier point est également étayé par l'intervenant 5. Celui-ci a insisté sur les différences d'expériences entre les membres de l'équipe :

"Au moins ça permet de parler le même langage que les autres membres de l'équipe qui n'ont pas le même poste ou background que nous".

Par ailleurs, les participants ont reconnu l'importance de justifier les choix utilisés pour l'Ubiquitous Language, afin de garantir sa cohérence et sa pertinence par rapport aux objectifs visés. Par exemple, selon l'intervenant 5, :

"Ils essayent d'utiliser des termes métiers afin de nous mettre directement dans le bain. Ils y intègrent un maximum de justifications afin de conserver la pertinence du terme utilisé par rapport à sa signification métier".

Ces justifications font l'objet d'une importante préparation en amont du projet comme l'ont souligné plusieurs intervenants et en particulier les analystes du projet :

"Heureusement, nous avons fait pas mal de réunion en amont des analyses. Ça a permis de pas mal démystifier les concepts et nous a aidé à définir les premières intentions métier" (intervenant 4).

Ils ont également mentionné que la démarche apportait une plus-value sur le long terme, comme l'indique l'extrait de l'intervenant 5 :

"En tant normal je dirais que ce n'est pas nécessaire ou obligatoire, mais plus de 2 ans après le début du projet je reconnais la plus-value qu'il y a derrière".

Néanmoins, il est nécessaire de nuancer les précédents propos, la démarche de l'Ubiquitous Langage n'étant pas toujours parfaite ni exhaustive, comme le souligne l'intervenant 5 :

"Oui, en tout cas pour les termes les plus généraux. Il nous manque surement certaines clés de compréhension pour des notions très particulières du métier d'huissier, mais je pense que dans l'ensemble, on ne doit pas trop mal se débrouiller".

6.1.2 Analystes

Du point de vue des analystes, nous retrouvons l'idée d'une préparation longue et complète en amont du projet. En effet, l'intervenant 4 explique :

"De base, je pense que nous avons eu presque 4 mois, à raison de 2-3 workshops par semaine pour arriver à capter un maximum d'infos".

Cette phase de pré-analyse est composée de plusieurs itérations avec le métier afin de capter au moins les intérêts de celui-ci, comme souligné à nouveau par l'intervenant 4 :

"Entre ces réunions, nous avons essayé de formaliser une première ébauche d'analyse et nous l'avons présenté régulièrement aux experts métiers afin d'avoir leur ressenti, être certain que nous allions dans la bonne direction".

L'objectif étant toujours de se conformer un maximum au vocabulaire du métier et à leurs besoins :

"En général on essaie de tacler un maximum de points lors de nos réunions d'analyse, on pose vraiment beaucoup de questions pour coller un maximum avec leurs besoins" (intervenant 4).

6.1.3 Développeurs

Selon les développeurs du projet, l'utilisation de ce langage commun permet une compréhension facilitée du code pour tous les membres de l'équipe, même lorsque l'analyse est complexe.

"Tout le monde est gagnant, car je ne me perds pas dans l'analyse, même si celle-ci est assez conséquente, et l'analyste peut comprendre une partie du code étant donné que les noms des objets viennent de lui" (intervenant 3).

D'autre part, l'utilisation des termes métiers directement dans le code permet d'améliorer la compréhension de ces termes au fil du temps :

"Mais comme on essaie d'utiliser les termes métiers directement dans notre code, à force de les utiliser, notre compréhension s'améliore, je pense" (intervenant 5).

Ainsi, l'utilisation d'un Ubiquitous Language facilite la compréhension et la communication au sein de l'équipe de développement, tout en permettant une meilleure appropriation du vocabulaire métier.

6.1.4 Ubiquitous Language et communication d'équipe

Dans cette section, nous souhaitons mettre en évidence les mécaniques propres à l'Ubiquitous Language ayant eu une répercussion positive, ou négative, sur la communication entre les membres de l'équipe et ayant ainsi influencé la compréhension du domaine métier.

De manière générale il est apparu que l'utilisation de l'Ubiquitous Language a grandement amélioré la communication entre les membres, principalement entre les analystes et les développeurs.

L'intervenant 4 a souligné l'importance de la cohérence linguistique entre tous les membres de l'équipe, ce qui concorde avec l'objectif principal évoqué par Evans dans son ouvrage de référence sur le DDD. Cette cohérence linguistique participe activement à la bonne communication d'équipe et reste un des principaux défis liés à la complexité des projets informatiques et à la diversité des compétences des intervenants comme nous l'avons déjà souligné dans notre revue littéraire. Néanmoins, ces contraintes en valent la peine comme le souligne Espinosa & al dans son article[83] : il y précise l'importance de la communication et de la collaboration au sein des équipes de projet informatique et la plus-value que celle-ci peut apporter. On remarque d'ailleurs que les intervenants 3 et 5 s'accordent sur cette bonne communication inter équipe malgré les différences de fonction ou d'expériences précédentes.

Un autre point faisant l'unanimité est l'utilisation des termes métiers afin de constituer le langage commun de l'équipe. Les analystes ont rencontré de manière régulières les experts métiers afin de constituer celui-ci. Nous savons déjà qu'il s'agit d'une bonne pratique issue des méthodes agiles et qu'elle influence positivement les modèles mentaux de l'équipe comme expliqué dans le point 3.3.1 (nous en reparlerons dans la section traitant de la combinaison des deux approches). Nous constatons en plus que l'utilisation des termes métiers dans les analyses a fortement aidé la compréhension du métier par les développeurs. Celle-ci, de leur propre aveu, n'est pas exhaustive en tout point, mais a eu le mérite de faciliter le dialogue au sein de l'équipe et apporter une plus-value certaine sur le long terme. Néanmoins, certains risques existent comme le soulèvent les analystes. En effet, les termes choisis pour constituer le langage commun doivent l'être avec soin et toujours justifiés dans leur contexte. Dans le cas contraire, cette mauvaise compréhension pourrait avoir des impacts directs sur la communication de l'équipe et le développement de l'application. Prendre le temps et demander des retours réguliers aux expert métiers devient donc une nécessité si l'on ne veut pas mettre à mal le langage commun.

Enfin, on pourrait se poser la question de la plus-value d'un tel langage pour les développeurs. Comme nous l'avons vu dans le point 3.2.1, l'une des principales difficultés rencontrée est la compréhension du code d'autrui et de facto sa capacité à travailler sur celui-ci. Plusieurs pistes sont déjà évoquées dans l'article de Latozza et al[35] afin de facilité le travail des développeurs et nous pouvons affirmer suite aux entretiens que le langage commun en fait partie. En plus de faciliter les interaction directe entre les développeurs et les analystes, le DDD préconise d'utiliser les termes métiers directement dans le code.

```
public class Debt {
    private double amount;
    private String debtorName;
    private String creditorName;
    private Date dueDate;
}

public class FormalNotice {
```

```
    private Debt debt;
    private Date issueDate;
    private String content;
}

public class LegalService {
    public FormalNotice createAndSendFormalNotice(Debt debt,
       String content) {
        // Check if the debt is overdue
        if (isDebtOverdue(debt)) {
            // Create the formal notice
            FormalNotice formalNotice = new FormalNotice(debt,
               new Date(), content);

            // Send the formal notice to the debtor
            sendFormalNoticeToDebtor(formalNotice);

            return formalNotice;
        }
        return null;
    }

    private boolean isDebtOverdue(Debt debt) {
        return debt.getDueDate().before(new Date());
    }

    private void sendFormalNoticeToDebtor(FormalNotice
       formalNotice) {
        // Logic to send the formal notice to the debtor
    }
}
```

Dans cet exemple, nous avons défini trois classes : *'Debt'* pour représenter une créance, *'FormalNotice'* pour représenter une mise en demeure, et *'LegalService'* pour gérer les actions légales liées aux créances. La méthode *'createAndSendFormalNotice'* dans la classe *'LegalService'* illustre le processus de création d'une créance et d'envoi d'une mise en demeure.

6.1.5 Impact sur l'évolution des modèles mentaux partagés

Dans la section précédente, nous avons établi l'importance de l'Ubiquitous Language pour améliorer la communication entre les membres de l'équipe et faciliter la compréhension du domaine métier. Il est maintenant temps d'explorer comment l'Ubiquitous Language peut influencer les modèles mentaux partagés au sein de l'équipe.

Les modèles mentaux partagés sont des représentations internes communes du monde et de ses éléments, permettant aux individus de comprendre et d'interpréter les actions et les intentions des autres membres de l'équipe[33]. La communication efficace et la compréhension du domaine métier sont essentielles pour développer et maintenir ces modèles mentaux partagés[84].

L'Ubiquitous Language joue un rôle central dans la formation et l'évolution des modèles mentaux partagés, en offrant un langage commun pour décrire et comprendre les concepts et les processus du domaine métier. En utilisant l'Ubiquitous Language, les membres de l'équipe peuvent aligner leurs représentations internes du domaine, favorisant ainsi la création et le renforcement de modèles mentaux partagés[85].

Par exemple, les analystes et les développeurs ont rapporté une meilleure compréhension du domaine métier grâce à l'utilisation des termes métiers dans le langage commun. Cette compréhension accrue facilite la convergence des modèles mentaux, ce qui permet à l'équipe de travailler de manière plus cohérente et harmonieuse[14]. De plus, l'utilisation des termes métiers directement dans le code, comme préconisé par le DDD, aide les développeurs à mieux comprendre le code d'autrui et à travailler plus efficacement ensemble.

Cependant, il est important de noter que la mise en place d'un langage commun ne garantit pas automatiquement la formation de modèles mentaux partagés. Les termes choisis pour constituer le langage commun doivent être sélectionnés avec soin et justifiés dans leur contexte. Les retours réguliers des experts métiers sont également essentiels pour maintenir et affiner l'Ubiquitous Language et, par conséquent, les modèles mentaux partagés[33]. Il convient de surveiller attentivement la communication et les interactions au sein de l'équipe pour détecter et résoudre les problèmes éventuels liés à la compréhension mutuelle et aux modèles mentaux.

En conclusion, l'Ubiquitous Language joue un rôle crucial dans la formation et le maintien des modèles mentaux partagés au sein d'une équipe. En offrant un langage commun pour décrire et comprendre les concepts du domaine métier, il facilite la communication et la compréhension mutuelle, favorisant ainsi la convergence des représentations internes et le développement de modèles mentaux partagés. Cependant, il est essentiel de sélectionner avec soin les termes constituant l'Ubiquitous Language et de solliciter régulièrement les retours des experts métiers pour maintenir et affiner ces modèles mentaux. En fin de compte, l'adoption de l'Ubiquitous Language, associée à une communication efficace et à une collaboration étroite entre les membres de l'équipe, peut contribuer à améliorer la performance de l'équipe et la qualité des projets informatiques.

6.2 Bounded Context et Context Map pour une meilleure organisation

Dans cette section, nous explorons les Bounded Contexts et les Context Maps, deux concepts-clé dans la compréhension du métier. Ces concepts sont définis et formalisés par les analystes lors d'une première phase conceptuelle avant d'être intégrés aux développements de l'équipe. Les Bounded Contexts servent à délimiter notre compréhension du métier en établissant des frontières qui séparent les différents modèles de données, tandis que les Context Maps représentent les relations entre ces Bounded Contexts.

"When a large system is decomposed into parts, it is still not practical to expect one model to apply to everything. Divide the work among multiple Bounded Contexts, each with its own model, and develop a Context Map to record the relationships between models."

– Eric Evans[31]

Pour comprendre l'impact de l'utilisation de ces concepts sur la compréhension du métier, l'organisation et la formation de modèles mentaux partagés au sein de l'équipe, nous avons mené des entretiens avec différents intervenants et leur avons demandé leur ressenti quant à

l'utilisation de cette découpe et l'influence que celle-ci avait sur leur compréhension du métier (Q18 et Q19).

Nos discussions couvrent plusieurs aspects, notamment les points de vue des intervenants sur les liens entre Ubiquitous Language et la manière de nommer les différents contextes, l'importance de la découpe pour la compréhension générale de l'application, la création de modèles mentaux partagés, ainsi que la fragmentation des connaissances.

Nous avons ensuite analysé les retours des analystes concernant l'importance de la découpe granulaire pour une meilleure structure et organisation des données, et comment cela favorise la formation de modèles mentaux partagés. Nous avons également abordé la relation entre la gestion des Bounded Contexts et l'architecture micro-services, ainsi que les avantages de cette approche pour la communication et la collaboration entre les équipes.

Du côté des développeurs, nous avons examiné leurs perspectives sur la mise en œuvre des Bounded Contexts et la structure du code en fonction de ces contextes, avec un accent particulier sur la réduction de la complexité, la facilitation de la maintenance et l'évolution de l'application.

Enfin, nous avons étudié les avantages spécifiques pour les développeurs les plus jeunes, tels que l'apprentissage d'une meilleure structure du code et du travail, la prévention des ambiguïtés et la participation à la création de modèles mentaux partagés pour une meilleure collaboration et une compréhension mutuelle.

6.2.1 Généralités

On peut d'ores et déjà souligner le lien que font certains intervenants entre la notion d'Ubiquitous Langage et la manière de nommer les différents contextes de l'application. En effet, l'intervenant 3 souligne :

"Oui, déjà rien que le fait de nommer le module en fonction du métier, de faire des frontières très clairs ça aide beaucoup à la compréhension générale de l'application."

Cette première compréhension de l'application, ainsi que du métier, est soulignée à plusieurs reprises par les intervenants. Les Bounded Context et les Context Map servant à délimiter les frontières de l'application, celles-ci sont représentées tant dans l'analyse que dans le code et sont souvent synonymes d'une première interaction avec les concepts métiers pour l'équipe, comme le souligne l'intervenant 4 :

"Au début du projet, c'était même hyper important, car ça m'a permis de comprendre les premiers concepts du métier."

De plus il apparaît évident pour l'équipe que cette division leur permet une meilleure vue haut niveau de l'application. Les Bounded Context étant des zones délimitant les concepts métiers et les Context Map permettant de représenter les liens concrets en ces zones, les intervenants font valoir la facilité avec laquelle il est possible de naviguer entre les concepts malgré leurs complexités.

"Ça nous permet de garder une vue high level de l'application, ou en tout cas du module sur lequel on est en train de travailler. On reste, je trouve, concentré sur l'élément sur lequel on travaille et c'est pour moi un peu plus facile dans un projet de cette envergure-là, de ne pas se confondre d'un élément à l'autre." (intervenant 2)

"Maintenant, c'est vrai que la façon dont on a découpé les choses permet d'aller relativement vite dans le ciblage." (intervenant 1)

Eric Evans souligne dans son ouvrage sur le DDD[31] la notion d'autonomie des équipes : en effet, en délimitant les responsabilités de chaque équipe dans le système, les Bounded Contexts permettent à chaque équipe de travailler de manière autonome et de se concentrer sur leur propre domaine. Cet aspect est également soulevé par l'équipe : il n'est pas toujours nécessaire d'avoir une compréhension exhaustive des modules de l'application pour être capable de travailler sur l'un d'eux, comme le précise l'intervenant 1 :

"Elle est fragmentaire, mais ça, cela vient du fait que justement, on a cassé la complexité de l'ensemble de l'application en divisant en différents domaines et qu'on n'a pas nécessairement demandé à tout le monde de connaître tous les domaines. Et donc ça permet de pouvoir s'investir dans un domaine sans nécessairement avoir la connaissance de l'entièreté des domaines de l'application."

Malgré les aspects positifs relevés par l'équipe, il peut néanmoins arriver que cette division métier pose des problèmes et peut parfois susciter plus de questions que de réponses :

"Je ne sais pas si ça aide toujours dans la compréhension du métier parce que des fois ça empêche parfois de faire des liens, puisqu'on développe vraiment de façon à en faire un Bounded Context un peu indépendant." (intervenant 1)

6.2.2 Analystes

Les analystes considèrent qu'il est important de découper les développements en différentes parties de manière précise dès le début. Cette découpe granulaire permet de mieux structurer et identifier les données qui se trouvent dans les différents contextes de l'application. L'intervenant 4 explique que cette approche leur permettait de mieux comprendre le modèle de domaine et de mieux organiser les différentes parties de l'application, en particulier la gestion des données dans chaque contexte :

"Ça permet vraiment de structurer les idées et de se concentrer que sur certaines parties qui sont indépendantes l'une de l'autre pour pouvoir mieux gérer et mieux structurer toutes les données en fonction du Bounded Context auquel ça appartient."

De plus, l'intervenant 1 souligne le parallèle entre la gestion des Bounded Context et l'architecture micro-services. En effet, la découpe en Bounded Contexts permet de définir des modules autonomes et indépendants, qui peuvent être implémentés comme des micro-services. Cette approche permet une gestion très indépendante des différents modules, qui peuvent communiquer avec des dossiers et des tiers, mais ne sont pas forcément interconnectés.

"Bien que les modules périphérique communiquent avec dossier et avec tiers, ils ne communiquent pas nécessairement entre eux. La découpe faite sur les différentes fonctionnalités de l'application pousse à une architecture micro-service et à une gestion très indépendante des différents modules."

6.2.3 Développeurs

Une fois que les Bounded Contexts ont été identifiés par les analystes, les développeurs peuvent les implémenter de différentes manières en fonction de l'architecture choisie. En général, chaque Bounded Context peut être implémenté comme un module indépendant, qui peut communiquer avec d'autres modules via des interfaces clairement définies et à l'aide d'évènement asynchrones, comme le souligne l'intervenant 5 :

"Il y a bien sûr des liens entre les différents contextes via une gestion d'évènements asynchrones, mais c'est justement pour respecter au mieux la découpe de l'application.".

En DDD, chaque Bounded Context représente une partie spécifique du domaine métier de l'application, avec des règles et des comportements propres à ce contexte. Les développeurs peuvent donc structurer leur code en fonction de chaque contexte, en se concentrant sur les fonctionnalités spécifiques à chaque contexte.

En structurant leur code de cette manière, les développeurs peuvent également réduire la complexité de l'application en isolant les différentes parties de l'application et en minimisant les dépendances entre elles. Cela permet également de faciliter la maintenance et l'évolution de l'application, car chaque Bounded Context peut être modifié indépendamment des autres.

C'est d'ailleurs ce que relève les plus jeunes développeurs en ce qui concerne la facilité d'apprendre à mieux structurer leur code et leur travail, mais aussi à éviter les ambiguïtés, comme expliqué par l'intervenant 3 :

"On structure vraiment notre code en Bounded Context. Donc en lisant les analyses, on peut facilement savoir où tel fichier se trouvera. On sait toujours clairement sur quoi on travaille, il n'y a pas d'ambiguïté.

6.2.4 Ubiquitous Language et compréhension des contextes

Comme nous l'avons vu dans la section précédente, l'Ubiquitous Language influence fortement les membres de l'équipe, leur compréhension du domaine métier et par extension les modèles mentaux partagés. Nous allons maintenant démontrer qu'il a un impact sur les autres artefacts du DDD et plus particulièrement sur les contextes.

L'utilisation de l'Ubiquitous Language a un impact significatif sur la compréhension des contextes par les membres de l'équipe. Lors des entretiens, l'intervenant 3 souligne que le fait de nommer les modules en fonction du métier et de définir des frontières claires facilite la compréhension générale de l'application. L'intervenant 4 ajoute que cette approche lui a permis de comprendre les premiers concepts du métier au début du projet.

Selon Björnson et Bratteteig[86], l'utilisation d'un langage commun au sein de l'équipe de développement permet de faciliter la compréhension des concepts métier et de réduire les erreurs liées aux ambiguïtés dans la communication. L'idée d'utiliser un langage spécifique au domaine (DSL) pour faciliter la communication et la compréhension partagée au sein des équipes de développement est en accord avec le concept de Bounded Context dans le DDD. Le Bounded Context met en avant l'importance de délimiter clairement les frontières d'un domaine spécifique, en utilisant l'Ubiquitous Language. En adoptant un langage commun basé sur les concepts métier, l'équipe peut mieux se concentrer sur les éléments spécifiques sur lesquels elle travaille. Cet aspect a été notamment été soulevé par l'intervenant 2.

En outre, Remy et Zayni[74] ont souligné que l'adoption d'un langage commun basé sur les concepts métier facilite la navigation entre les concepts, malgré leur complexité. Selon Remy et Zayni, l'utilisation de DSL permet de modéliser les concepts métier et leurs relations de manière plus expressive, en évitant la complexité inhérente aux langages de programmation généraux. Cela facilite non seulement la communication et la collaboration entre les membres de l'équipe, mais également la compréhension des concepts métier et des règles associées.

Dans le cadre du DDD, l'Ubiquitous Language est étroitement lié à la notion de Bounded Context. En nommant les Bounded Contexts en fonction des concepts métier, il devient plus facile pour les membres de l'équipe de comprendre les responsabilités de chaque contexte et d'interagir avec eux de manière cohérente. Alcides et Pereira[87] affirment que l'utilisation d'un langage commun basé sur les concepts métier favorise la cohésion et la modularité de l'application, ce qui facilite la maintenance et l'évolutivité du système.

En résumé, l'adoption d'un Ubiquitous Language basé sur les concepts métier permet d'améliorer la compréhension des contextes par les membres de l'équipe de développement, en facilitant la communication, en réduisant les ambiguïtés et en favorisant l'alignement des modèles mentaux partagés.

6.2.5 Rôle des analystes dans la définition des Bounded Contexts

Le rôle des analystes est crucial pour la définition des Bounded Contexts au début d'un projet. Ils sont responsables de la compréhension du domaine métier et de la traduction de cette compréhension en termes techniques, en définissant les frontières des Bounded Contexts. Cette première étape permet de structurer l'application et d'organiser les données en fonction des différents contextes métiers.

Dans les entretiens réalisés, l'intervenant 4 explique que cette approche permet de mieux comprendre le modèle de domaine et d'organiser les différentes parties de l'application, en particulier la gestion des données dans chaque contexte.

Alcides et Pereira confirment également l'importance du rôle des analystes dans la définition des Bounded Contexts. Dans leur étude, ils soulignent que les analystes sont chargés de comprendre le domaine métier, d'identifier les concepts clés et les règles, et de déterminer les frontières entre les différents Bounded Contexts en fonction de ces concepts et règles [87].

En outre, Remy et Zayni mettent en avant l'importance des DSL dans la conception pilotée par le domaine, en soulignant que les analystes peuvent utiliser ces langages pour décrire le domaine métier et faciliter la communication entre les parties prenantes[74]. Cela peut aider les analystes à définir les Bounded Contexts et à faciliter la compréhension du domaine par l'ensemble de l'équipe.

La définition des Bounded Contexts par les analystes est également importante pour faciliter la collaboration entre les équipes et les membres de l'équipe, comme le souligne l'intervenant 1 dans l'extrait 6.2.2.

Nous pouvons donc affirmer que les analystes jouent un rôle clé dans la définition des Bounded Contexts, en traduisant leur compréhension du domaine métier en termes techniques et en structurant l'application en fonction des différents contextes métiers. Cette étape est cruciale pour faciliter la compréhension du domaine par l'ensemble de l'équipe et favoriser l'établissement de modèles mentaux partagés. Ainsi, en organisant les données et en encourageant la collaboration et la communication entre les membres de l'équipe, les analystes contribuent à renforcer l'alignement autour d'une vision commune et à améliorer la cohésion du projet.

6.2.6 Mise en œuvre des Bounded Contexts par les développeurs

La mise en œuvre des Bounded Contexts par les développeurs consiste à structurer le code en fonction des contextes définis par les analystes. Cette approche a plusieurs avantages, tels que la réduction de la complexité[31], la facilitation de la maintenance et l'évolution de l'application[88], ainsi que le développement de modèles mentaux partagés au sein de l'équipe[87].

Dans les entretiens réalisés, l'intervenant 5 explique que la mise en œuvre des Bounded Contexts permet de simplifier le code et d'améliorer sa lisibilité grâce à une meilleure isolation des parties du code et une simplification des interactions entre les différents composants.

Un exemple concret de la mise en œuvre des Bounded Contexts dans le code pourrait être l'utilisation de répertoires séparés pour chaque contexte, avec des noms de répertoires clairs

et explicites[71]. Ainsi, les développeurs peuvent facilement identifier les parties du code qui appartiennent à un contexte spécifique et éviter les confusions.

```
BailiffManagementApplication/
├── clientManagementContext/
│   ├── clientRegistration/
│   │   ├── models/
│   │   ├── services/
│   │   └── repositories/
│   └── clientBilling/
│       ├── models/
│       ├── services/
│       └── repositories/
├── debtRecoveryContext/
│   ├── debtCaseManagement/
│   │   ├── models/
│   │   ├── services/
│   │   └── repositories/
│   └── legalProceedings/
│       ├── models/
│       ├── services/
│       └── repositories/
└── documentManagementContext/
    ├── templates/
    ├── documentGeneration/
    └── documentStorage/
```

Dans cet exemple, nous avons trois Bounded Contexts : *'clientManagementContext'*, *'debtRecoveryContext'*, et *'documentManagementContext'*. Chacun d'eux contient des sous-répertoires spécifiques à leur domaine métier, tels que *'clientRegistration'*, *'clientBilling'*, *'debtCaseManagement'*, *'legalProceedings'*, *'templates'*, *'documentGeneration'*, et *'documentStorage'*. Chaque sous-répertoire contient des dossiers pour les modèles, services et dépôts, lorsque cela est pertinent.

Milovidov[89] discute des défis pratiques liés à la mise en œuvre du DDD et souligne l'importance de la collaboration étroite entre les analystes et les développeurs pour assurer une compréhension partagée du domaine métier et faciliter la communication entre les parties prenantes.

L'intervenant 3 souligne également l'importance de la mise en œuvre des Bounded Contexts pour la collaboration et la communication entre les équipes et les membres de l'équipe :

"Le fait d'avoir des Bounded Contexts clairement définis et implémentés facilite grandement la communication et la collaboration entre les équipes, car chacun sait quelles sont ses responsabilités et comment interagir avec les autres contextes."

En somme, la mise en œuvre des Bounded Contexts par les développeurs est cruciale pour structurer le code, réduire la complexité, faciliter la maintenance et l'évolution de l'application. Cette approche favorise également la collaboration et la communication entre les équipes et les membres de l'équipe, en clarifiant les responsabilités et les interactions entre les différents contextes.

6.2.7 Avantages pour les développeurs les plus jeunes

Les entretiens réalisés avec les intervenants ont révélé plusieurs avantages spécifiques pour les développeurs en début de carrière lors de l'utilisation des Bounded Contexts et des Context Maps dans le développement de logiciels.

Les concepts de Bounded Context et Context Map aident les jeunes développeurs à apprendre à structurer correctement le code et à organiser leur travail de manière efficace. En travaillant avec des contextes clairement définis, ils peuvent développer des compétences en matière de modularité et de réduction de la complexité[75]. L'intervenant 2 a mentionné :

"Lorsque j'ai commencé à travailler avec des Bounded Contexts, j'ai rapidement réalisé comment cela m'a aidé à organiser mon code de manière plus logique et efficace."

L'adoption de l'Ubiquitous Language et la définition précise des Bounded Contexts contribuent à éliminer les ambiguïtés dans le code et la communication entre les membres de l'équipe[74]. Un développeur débutant (intervenant 3) a déclaré :

"En utilisant un langage commun et en comprenant clairement les frontières de chaque contexte, j'ai pu me concentrer sur la résolution de problèmes spécifiques sans me soucier des malentendus."

Enfin, les développeurs débutants peuvent participer activement à la création de ces modèles mentaux partagés, améliorant ainsi leur compréhension du domaine métier et de l'architecture logicielle. L'intervenant 5 a mentionné :

"Les développeurs les moins expérimentés ont pu contribuer et comprendre notre architecture et notre domaine grâce aux Bounded Contexts clairement définis."

6.2.8 Contextes et modèles mentaux partagés

En conclusion, ce chapitre a mis en évidence l'importance de la classification en Bounded Context et Context Map pour une meilleure organisation, une compréhension plus profonde du domaine métier et la formation de modèles mentaux partagés au sein des équipes de développement. Les concepts de Bounded Context et Context Map, couplés à l'adoption de l'Ubiquitous Language, permettent de définir clairement les frontières et les responsabilités, facilitant ainsi la collaboration entre les membres de l'équipe et améliorant la qualité du code.

Les discussions avec les différents intervenants ont souligné l'impact de cette approche sur la compréhension du domaine, la structure et l'organisation des données, ainsi que la communication entre les équipes. Les analystes jouent un rôle clé dans la définition des Bounded Contexts, tandis que les développeurs sont responsables de leur mise en œuvre et de la gestion de la complexité du code. Les avantages spécifiques pour les développeurs en début de carrière ont également été examinés, mettant en évidence la valeur ajoutée de cette approche pour leur formation et leur intégration au sein de l'équipe.

En somme, la classification en Bounded Context et Context Map constitue une approche essentielle pour les équipes de développement qui cherchent à améliorer leur organisation, leur compréhension du domaine métier et leur collaboration. Elle permet de favoriser la création de modèles mentaux partagés et de renforcer la communication entre les différentes parties prenantes.

6.3 Les Tactical Patterns pour une compréhension améliorée

Les Tactical Patterns, conçus pour traiter les aspects techniques de l'implémentation de l'architecture de domaine, sont des modèles de conception opérationnels qui s'attaquent à des

problèmes précis comme la gestion des transactions, la division des responsabilités, et la modélisation d'Aggregate, de services et d'événements. Ces concepts, introduits par Eric Evans dans le DDD, sont indispensables pour établir une compréhension claire du domaine et faciliter la communication au sein de l'équipe.

"Use the most abstract elements, the Tactical Patterns, to create simple models of the design that can be understood and manipulated as a whole. This includes the high-level relationships between objects in the domain model, along with the major responsibilities and collaborations."

– Eric Evans[31]

Typiquement, ces modèles sont mis en œuvre dans un ou plusieurs Bounded Contexts, qui sont des sous-systèmes spécifiques d'un système plus large, ayant leurs propres règles et modèles métier distincts. Dans notre étude de cas, l'application des Tactical Patterns est généralisée à toute l'équipe. En effet, ce sont les analystes qui déterminent les différentes Entities, Aggregate, services, etc., alors que les développeurs sont chargés de la mise en œuvre de ces directives.

Notre enquête porte sur l'aspect général de la compréhension métier via l'utilisation des Tactical Patterns (Q16 et Q17), sur la valeur ajoutée par les analystes en définissant ces Entities (Q34 et Q35), et sur l'avis des développeurs concernant la découpe réalisée et sa pertinence (Q40 et Q41).

Nos discussions se déclinent en plusieurs sections, en fonction des retours d'entretien et des observations. Ces sections concernent la compréhension globale de l'application et la hiérarchisation des artefacts, l'impact des Tactical Patterns sur les analystes et les développeurs, ainsi que les bénéfices pour les développeurs les plus juniors.

Nous examinerons comment la structuration des artefacts via les Tactical Patterns facilite la compréhension générale de l'application et aide les membres de l'équipe à identifier les rôles métier des divers éléments. Ensuite, nous aborderons comment cette approche impacte les analystes, en soulignant comment elle améliore la collaboration entre eux et les développeurs. Enfin, nous analyserons les avantages des Tactical Patterns pour les développeurs débutants, en montrant comment cette méthode leur permet d'acquérir des compétences et de progresser plus rapidement dans leur carrière.

6.3.1 Généralités

Un aspect essentiel régulièrement évoqué par les intervenants concerne la hiérarchisation des différents artefacts du Tactical Pattern et la facilité que cela apporte à la compréhension globale de l'application. En effet, l'équipe met en place une série d'Aggregates Root, Entity, Value Object, etc. possédant chacun des caractéristiques fortes permettant une meilleure identification de leur rôle métier, comme l'explique l'intervenant 2 :

"En partie dans le sens où le fait d'avoir justement des Value Object, des Aggregate, Entities, etc ; permet d'avoir une meilleure vision de l'importance des différents éléments."

L'Aggregate Root étant l'unique point d'entrée pour accéder aux données relatives à l'Aggregate, celui-ci tend à refléter la découpe préalable des Bounded Context pour les objets les plus importants, comme le souligne l'intervenant 3 :

"C'est comme avec les Aggregates Root. Comme ils collent à la découpe du projet, tu sais toujours où effectuer tes modifications et ce que cela va impacter."

Dans le cas de l'application développée par l'équipe que nous suivons, il s'avère que les Bounded Context principaux s'articulent autour de la notion de 'Dossier', 'Tiers', 'Documents',

'Bilan', etc. Pour chaque module, un Aggregate Root du même nom est défini comme point d'entrée de celui-ci. Les artefacts enfants de l'Aggregate seront généralement des Entity ou des Value Object, ce qui n'empêche pas pour autant de retrouver d'autres Aggregates Root dans cette sous-structure.

Suivant généralement les Aggregates Root, nous retrouvons dans la structure générale de l'application les artefacts Entity ou encore Value Object. Ces deux objets ayant des caractéristiques et comportements différents, comme la présence d'un identifiant pour les Entity ou la mutabilité des Value Object, la distinction entre les deux ainsi que leurs attributions à des concepts métiers permettent à l'équipe de mieux cerner ceux-ci :

"Comme le sujet est déjà assez complexe, le fait de découper tout en Value Object ou Entity par exemple nous aide déjà à comprendre l'utilité de ces objets." (intervenant 3).

"Si nous avons à faire à une entity, nous savons qu'elle aura forcément un Id et que celle-ci sera mutable. Contrairement à la Value Object qui sera immutable et remplacée à chaque édition." (intervenant 5)

Un problème récurent dans le développement logiciel concerne les désalignements potentiels entre les analyses et le code développé[90]. En effet, il n'est pas rare de voir les développeurs prendre des libertés quant aux analyses rédigées par leurs collègues pour diverses raisons. Néanmoins, cela semble différent dans le DDD, comme le précise l'intervenant 3 :

"Étant donné que ce sont les analystes qui définissent les éléments du Tactical Pattern ou encore les frontières des Bounded Context, et que nous, développeurs, les gardons tels quels dans le code, on sait directement de quoi on parle quand il y a des questions ou un souci."

6.3.2 Analystes

Malgré les aspects plus techniques du Tactical Pattern, les analystes soulignent la facilité d'utilisation de celui-ci et la plus-value qu'il peut apporter aux analyses quand il est totalement maîtrisé. Néanmoins, ils rappellent que la connaissance du métier du client reste essentielle pour l'application du Tactical Pattern, souligné par l'intervenant 5 :

"Si on arrive à bien cerner les concepts métiers les plus importants, il devient facile d'utiliser le Tactical Pattern à bon escient."

Le lead analyste (intervenant 1) illustre les précédents propos à l'aide d'une discussion tenue avec un analyste junior (intervenant 4) :

"Par exemple, j'ai demandé il y a quelques semaines à X : "il faut que tu ailles modifier un Event, telle Entity sous telle condition... Tu me pop un Event qui s'appelle comme ça...." Je n'ai même pas dû lui expliquer pourquoi, mais il est allé dans la bonne colonne. Donc, il savait où il fallait aller pour le faire. Et il n'avait pas nécessairement besoin d'avoir la connaissance de ce qui se passait après que l'Event a été généré."

6.3.3 Développeurs

À l'instar des Bounded Context, les développeurs implémentent les artefacts du Tactical Pattern à la suite de la documentation produite par les analystes. Nous remarquons que les plus jeunes développeurs insistent sur la structure du code que leur impose l'utilisation du Pattern et de la plus-value que celui-ci confère par rapport à l'apprentissage scolaire. L'intervenant 3 explique :

"En tout cas ça m'aide à mieux structurer mon code. Le Tactical Pattern utilisant une hiérarchie entre les objets assez précise je peux plus facilement positionner ceux-ci dans mon projet. Ce n'est pas vraiment quelque chose que l'on apprend à l'école, ou en tout juste les grandes lignes sans rentrer dans une application pratique comme c'est le cas dans le monde du travail"

De plus, dans le contexte d'un développement en équipe, il n'est pas rare de voir les développeurs travailler ou reprendre les tâches d'un de leurs collègues. Les difficultés de compréhension du code d'autrui sont régulièrement mises en avant[35] comme un frein récurrent au sein de l'équipe, néanmoins l'intervenant 2 explique :

Quelqu'un d'extérieur ne devrait pas avoir de difficulté à rentrer dans le code, pour peu qu'on lui explique le fonctionnement de l'approche mise en place.

ou encore de la part de l'intervenant 5, le plus expérimenté de l'équipe :

"Oui ça devient très facile ! On ne travaille pas tous sur les mêmes epics et parfois il nous arrive de devoir effectuer une tâche sur une partie du code que l'on découvre pour la première fois. Bien sûr, il n'y a pas que le Tactical Pattern qui rentre en compte, mais ça n'en reste pas moins un facilitateur pour celui qui découvre le projet et le code produit par un développeur."

Encore une fois l'équipe nous rappelle le temps que peut prendre l'apprentissage du DDD avant de pouvoir l'appliquer de manière correcte, mais également la plus-value sur le long terme, comme le souligne l'intervenant 3 :

"Sans ça, c'est la catastrophe assurée, mais si tu prends le temps de bien comprendre toute l'approche du DDD et surtout du Tactical Pattern pour un dev ça devrait réellement te faciliter les choses. Mais je pense qu'il y aura quand même une plus-value sur la compréhension du code, donc je pense que ça équilibrera largement le temps perdu à apprendre le DDD."

Enfin une analyse rappelle la volonté de l'équipe d'avoir une compréhension commune du métier et pas seulement les analystes, seuls membres en contact direct avec le métier. Comme le souligne l'intervenant 4, le Tactical Pattern permet aux développeurs de se questionner sur l'utilisation de celui-ci dans le cadre de leur code :

"Nous voulons autant que possible que les devs comprennent les spécifications du métier et pas juste qu'ils suivent bêtement l'analyse."

6.3.4 Compréhension globale et hiérarchisation des artefacts

La compréhension globale de l'application et la hiérarchisation des artefacts sont des aspects essentiels dans le cadre du DDD. Selon Vernon et Nilsson[88, 71], la structure et l'organisation des artefacts du DDD permettent une meilleure compréhension du système et facilitent la communication au sein de l'équipe de développement. La mise en place d'une hiérarchie claire pour les objets tels que les Aggregate Roots, les Entities et les Value Objects renforce leur rôle dans le domaine métier et aide les membres de l'équipe à comprendre leur importance relative.

Dans les interviews réalisées, les participants soulignent l'importance de cette hiérarchisation pour la compréhension de l'application. Par exemple, l'intervenant 2 mentionne que l'utilisation de Value Objects, d'Entities et d'Aggregate facilite la distinction entre les différents éléments et leur rôle métier. De même, l'intervenant 3 explique que la correspondance entre les Aggregate Roots et la découpe du projet permet de mieux comprendre l'impact des modifications et de connaître les points d'entrée pour accéder aux données.

En outre, la structure des artefacts du DDD permet aux développeurs de mieux organiser leur code et d'identifier plus facilement les objets dans le projet. Comme le note l'intervenant 3,

cela contraste avec l'enseignement scolaire, qui ne fournit souvent que des connaissances théoriques sans montrer comment les appliquer de manière pratique. La hiérarchisation des artefacts du DDD facilite donc la compréhension globale de l'application et améliore la collaboration et la communication entre les membres de l'équipe de développement.

En intégrant les leçons tirées des interviews, il apparaît que les développeurs et les analystes trouvent un réel avantage à appliquer la hiérarchisation des artefacts dans leur travail quotidien. Cette approche permet une meilleure communication entre les membres de l'équipe et une meilleure compréhension des besoins métier. Cela montre l'importance de la hiérarchisation des artefacts et de la compréhension globale de l'application dans le succès du DDD.

6.3.5 Alignement entre les analyses et le code développé

L'harmonisation entre les analyses et le code produit constitue un élément crucial du DDD, favorisant une meilleure appréhension du domaine métier et une communication performante au sein de l'équipe. Les entretiens menés soulignent que le Tactical Pattern soutient cet équilibre, garantissant une adéquation entre les composants déterminés par les analystes et leur mise en œuvre par les développeurs.

Un intervenant mentionne que l'alignement entre les analyses et le code développé est facilité par le fait que les développeurs suivent les artefacts définis par les analystes, permettant une meilleure communication et une résolution plus rapide des problèmes comme l'a expliqué l'intervenant 5 dans son interview 6.3.1.

D'après Stamelos et al.[90], un désalignement entre les analyses et le code développé peut provoquer des problèmes de compréhension et de maintenance du code, ainsi que des difficultés de communication entre les membres de l'équipe. L'application du Tactical Pattern permet aux équipes de surmonter ces problèmes en facilitant une meilleure organisation et structuration du code. En effet, le Tactical Pattern encourage la création de composants bien définis, tels que les Aggregate Roots, les Entities et les Value Objects, qui contribuent à rendre le code plus lisible et compréhensible.

De plus, le Tactical Pattern facilite la communication au sein de l'équipe en offrant un langage commun pour décrire les différents éléments du domaine métier. Ceci permet aux analystes et aux développeurs de partager la même vision du système, de mieux comprendre les besoins métier et les attentes des parties prenantes, et de développer des modèles mentaux partagés. Ces modèles mentaux partagés sont essentiels pour une communication efficace, car ils assurent que tous les membres de l'équipe comprennent et interprètent les informations de la même manière.

En conséquence, le processus de développement est plus fluide et les risques d'erreurs et de malentendus sont réduits. L'adoption du Tactical Pattern contribue à un alignement plus étroit entre les analyses et le code développé, favorisant une meilleure compréhension du domaine métier, des modèles mentaux partagés et une communication efficace entre les membres de l'équipe. Cela se traduit par une meilleure qualité du code et une maintenance plus aisée, conduisant à un développement plus efficace et performant.

Par ailleurs, Björnson et Bratteteig[86] mettent en avant l'importance de la collaboration entre les analystes et les développeurs pour faciliter la compréhension du domaine métier et la communication entre les parties prenantes. Le Tactical Pattern, en structurant les artefacts et en définissant clairement leur rôle métier, facilite cette collaboration et aide les équipes à maintenir l'alignement entre les analyses et le code développé.

Pour conclure, le Tactical Pattern contribue à l'alignement entre les analyses et le code développé en favorisant la cohérence des artefacts et en renforçant la collaboration entre les analystes et les développeurs. Cette approche facilite la compréhension du domaine métier et améliore la communication entre les membres de l'équipe, conduisant à une meilleure qualité du code et une maintenance plus aisée.

6.3.6 Influence du Tactical Pattern sur les analystes

L'impact du Tactical Pattern sur les analystes est un élément crucial du DDD. Les entretiens réalisés révèlent que la mise en œuvre du Tactical Pattern améliore la communication entre les analystes et les développeurs et permet aux analystes de saisir plus aisément les concepts métier.

Un analyste (intervenant 5) met en avant l'importance de bien maîtriser les concepts métier pour utiliser efficacement le Tactical Pattern dans l'extrait 6.3.2.

La recherche en génie logiciel souligne aussi l'importance de la communication entre les analystes et les développeurs pour garantir la qualité et le succès d'un projet[91]. Le Tactical Pattern, en clarifiant les artefacts et leur rôle métier, encourage cette communication et facilite la compréhension des concepts métier.

En outre, les analystes peuvent se servir du Tactical Pattern pour mieux organiser leur analyse et déterminer les limites des Bounded Contexts, ce qui simplifie le processus d'analyse et de conception. L'adoption de modèles mentaux partagés favorise une meilleure appréhension des concepts métier et rend la communication entre les analystes et les développeurs plus aisés. Le participant 1 illustre ce point en mentionnant une conversation avec un analyste junior (intervenant 4) dans le point 6.3.2.

Pour conclure, l'application du Tactical Pattern a un impact favorable sur les analystes en améliorant la communication avec les développeurs, en renforçant la compréhension des concepts métier grâce aux modèles mentaux partagés et en aidant à structurer l'analyse et la conception du logiciel. Cette approche accroît la qualité de l'analyse et contribue au succès global du projet.

6.3.7 Impact du Tactical Pattern sur les développeurs

L'impact du Tactical Pattern sur les développeurs est essentiel dans le cadre du DDD. Les entretiens menés révèlent que la mise en œuvre du Tactical Pattern favorise une organisation et une structure du code plus efficaces, ainsi qu'une meilleure appréhension des concepts métier par les développeurs.

L'intervenant 3 indique dans l'extrait 6.3.3 que le Tactical Pattern facilite la structuration du code et la hiérarchie des objets, aidant ainsi les développeurs à positionner plus aisément les différents éléments dans le projet.

Selon Naur et Randell[92], il est primordial de structurer et d'organiser le code efficacement pour en simplifier la compréhension et la maintenance. Le Tactical Pattern y contribue en établissant une hiérarchie claire des artefacts et en précisant leur rôle dans le domaine métier. Ainsi, les développeurs peuvent améliorer l'organisation de leur code, ce qui facilite l'appréhension des concepts métier et encourage l'adoption de modèles mentaux partagés au sein de l'équipe de développement. Cette approche coordonnée permet une collaboration optimale entre les membres de l'équipe et améliore la qualité du code produit.

Par ailleurs, Latoza et al.[93] soulignent que comprendre le code des autres est un enjeu majeur pour les développeurs travaillant en équipe. Le Tactical Pattern, en imposant une structure

claire et en définissant les responsabilités des artefacts, facilite la compréhension du code par les autres membres de l'équipe, réduisant ainsi les difficultés de collaboration et de maintenance.

En résumé, l'application du Tactical Pattern a un impact favorable sur les développeurs en améliorant la structure et l'organisation du code, en approfondissant la compréhension des concepts métier et en favorisant la collaboration entre les membres de l'équipe. Cette approche conduit à une meilleure qualité du code et à une maintenance plus simple, ce qui est bénéfique pour l'ensemble du projet.

6.3.8 Compréhension commune du métier

La compréhension mutuelle du domaine métier est un facteur clé de succès pour un projet DDD. Les entretiens montrent que le Tactical Pattern joue un rôle important pour encourager cette compréhension partagée parmi les membres de l'équipe.

L'intervenant 4 met en évidence l'importance pour les développeurs de bien saisir le domaine métier, en plus des analystes, et comment le Tactical Pattern peut les y aider :

"Nous souhaitons que les développeurs comprennent autant que possible les spécifications métier et ne se contentent pas de suivre aveuglément l'analyse."

Cela rejoint les travaux de Pichler et Pimentel[94], qui affirment que l'adéquation entre les connaissances métier et les compétences techniques est cruciale pour la réussite d'un projet. En utilisant le Tactical Pattern, les développeurs sont en mesure de mieux appréhender les concepts métier et de les intégrer dans leur code, améliorant ainsi la qualité du logiciel final.

Un autre aspect crucial de la compréhension partagée du domaine métier concerne la collaboration entre les membres de l'équipe. La littérature en génie logiciel met en avant l'importance de la communication et de la collaboration pour la réussite des projets[95]. Le Tactical Pattern, en précisant les rôles et responsabilités des artefacts, facilite une meilleure communication et collaboration entre les membres de l'équipe, renforçant ainsi la compréhension commune du domaine métier et, par conséquent, les modèles mentaux partagés.

6.3.9 Tactical Pattern et modèles mentaux partagés

En conclusion, le Tactical Pattern est essentiel pour l'organisation, la compréhension et la coopération au sein d'une équipe appliquant le DDD. Il offre une structure claire du code et une hiérarchie bien définie des artefacts, améliorant la compréhension globale de l'application pour tous les membres de l'équipe. De plus, il permet aux développeurs d'adhérer étroitement aux analyses réalisées par les analystes, réduisant ainsi les désalignements entre les spécifications et le code produit.

Le Tactical Pattern facilite également la structuration du code et approfondit la compréhension des concepts métier. Il soutient la collaboration et la communication entre les membres de l'équipe et a un impact positif sur les analystes en leur permettant de présenter les concepts métier de manière plus claire et précise. Cela facilite la communication avec les développeurs et contribue à la réussite du projet.

De plus, le Tactical Pattern encourage une compréhension partagée du domaine métier entre les membres de l'équipe en favorisant la communication et la collaboration. Il permet aux développeurs d'intégrer les concepts métier de manière plus cohérente dans leur code et stimule le développement de modèles mentaux partagés au sein de l'équipe.

Dans l'ensemble, le Tactical Pattern se révèle être un outil précieux pour les équipes utilisant le DDD, en promouvant une meilleure organisation, une compréhension partagée du domaine

métier et une collaboration efficace entre les membres de l'équipe. Les entretiens réalisés avec divers intervenants renforcent cette conclusion, mettant en évidence les avantages tangibles du Tactical Pattern dans un contexte réel de développement logiciel.

6.4 Les défis et opportunités du DDD

Le DDD, malgré son potentiel à traiter efficacement des domaines complexes, est souvent perçu comme une approche difficile à mettre en œuvre, comme le suggèrent Vernon et al.[88]. Plusieurs facteurs contribuent à cette complexité :

— L'abstraction de domaine : DDD implique la création d'une représentation à la fois complexe et abstraite du domaine métier, demandant une compréhension approfondie de ce dernier[96]. La difficulté se révèle particulièrement prégnante si les développeurs ne sont pas experts dans le domaine.

— La communication : La nécessité d'une collaboration étroite entre les experts du domaine et les développeurs afin d'assurer que le modèle de domaine reflète fidèlement la complexité du domaine métier est un autre défi de taille[97]. En particulier, si les experts du domaine et les développeurs sont géographiquement éloignés ou ont des horaires différents, la communication devient encore plus difficile[41, 23].

— La flexibilité : La capacité d'adaptation du modèle de domaine à une évolution constante du métier est essentielle[98]. Les défis peuvent se présenter lorsque le modèle de domaine est trop rigide ou si les développeurs ne comprennent pas suffisamment les besoins évolutifs du domaine.

— L'implémentation : La mise en place du DDD peut nécessiter des compétences techniques avancées et une expérience significative du développement logiciel, ajoutant à sa complexité. Selon Alhir[99], cette complexité est multiforme.

Cette complexité est ressentie à travers les réponses des participants à nos entretiens, notamment lorsqu'ils comparent leurs expériences actuelles et passées et lorsqu'ils répondent aux questions Q24, Q26 et Q27 sur les performances de l'équipe ainsi que les aspects généraux positifs et négatifs.

Pour surmonter ces défis, il est essentiel que les équipes travaillant avec le DDD adoptent une approche de conception flexible et axée sur l'évolution. En mettant l'accent sur la collaboration avec les experts du domaine, les développeurs peuvent acquérir une compréhension approfondie des besoins métier et de leur évolution. Les équipes doivent aussi être prêtes à revoir et adapter régulièrement leurs modèles de domaine pour refléter les changements dans le domaine métier.

L'usage de Tactical Patterns, tels que les Aggregate Roots, les Entities et les Value Objects, peut contribuer à favoriser une conception flexible en définissant clairement les rôles et les responsabilités des différents artefacts du domaine. De plus, l'implémentation du DDD exige une expertise technique pour assurer que le modèle de domaine est adéquatement intégré à l'application. Les développeurs doivent ainsi travailler en étroite collaboration avec les experts du domaine, appliquer des modèles et principes de conception spécifiques, et être prêts à adapter et évoluer leurs modèles de domaine à mesure que les besoins métier changent.

En résumé, la mise en œuvre du DDD est complexe en raison de la nécessité de combiner des compétences techniques avancées, une compréhension approfondie du domaine métier et une maîtrise des concepts et des outils spécifiques au DDD. Malgré ces défis, les équipes qui parviennent à mettre en œuvre le DDD efficacement sont mieux placées pour créer des logiciels flexibles et évolutifs qui répondent aux besoins changeants du domaine métier.

6.4.1 Généralités

Afin de contrer cette complexité les intervenants évoquent tous la nécessité d'avoir une connaissance exhaustive du DDD. Cet aspect est d'ailleurs illustré par l'intervenant 1 :

"Si tu ne comprends pas quelle est la différence entre une Value Object et une Entity par exemple au niveau de leur cycle d'évolution, ça va se transformer en catastrophe et la personne va vraiment avoir l'impression de subir l'analyse. Je pense qu'à partir du moment où tous les concepts du DDD sont clairs dans la tête des analystes, il n'y a pas de réel problème."

ou encore, venant de l'intervenant 2 :

"Je pense que si on le fait à moitié et qu'on s'autorise des largesses par rapport à certains grands principes du DDD, on va vite se retrouver à avoir un truc qui est moins clair que si on n'avait pas appliqué le DDD du tout."

De plus il est évident que cette connaissance de l'approche doit s'appliquer à l'ensemble de l'équipe du projet et non qu'à une partie d'entre eux, comme le souligne l'intervenant 2 :

"Mais à nouveau, c'est un investissement, car il faut que l'ensemble de l'équipe soit aligné sur cette approche et sur comment la mettre en application."

Comme souligné ci-dessus l'investissement de l'équipe doit être complet et cela ne se fait pas toujours avec facilité. Le lancement du projet a été une étape compliquée pour plusieurs membres de l'équipe comme l'explique l'intervenant 3 :

"Par contre on sent que c'était un peu plus laborieux au début du projet. Tout le monde n'avait pas, je pense bien compris la façon de fonctionner, moi le premier. On a dû me réexpliquer plusieurs fois le Tactical Pattern ou encore le langage commun des huissiers."

Enfin, il est important de préciser que l'utilisation du DDD n'est pas justifiable pour tous les types de projets. Une certaine complexité métier est nécessaire pour pouvoir appliquer l'approche de manière cohérente et efficace, comme le souligne l'intervenant 1 :

"On ne le ferait pas pour n'importe quel projet. Il faut un projet, je pense assez conséquent, avec assez d'éléments pour que ça en vaille la peine. Mais dans le cas d'une application complexe, avec plusieurs niveaux dans le métier, là ça devient vraiment justifié."

Néanmoins, malgré ces premiers obstacles tous les intervenants s'accordent à dire que l'application dans son ensemble gagne en stabilité et en évolutivité suite à la mise en œuvre du DDD. Ces points positifs sont mis en avant par les intervenants 1 et 4 :

"Quelle est le ratio ? la perte en termes de vélocité et de développement ? Et on était entre 1.5 et deux, deux fois plus lent avec cette méthode-là. Et ça, tout le monde était plus ou moins d'accord, c'est qu'on gagnerait énormément en évolutivité et en stabilité de l'application. C'est l'investissement de départ, il est difficile."

et

"Il y a beaucoup moins d'allers-retours, moins de bugs détectés, moins de questions de la part des devs."

6.4.2 Analystes

Les analystes insistent sur le temps nécessaire en amont pour préparer au mieux le projet, tant d'un point de vue technique que fonctionnel. En effet, il n'est pas rare de constater un démarrage en parallèle des analyses et du développement, ce qui peut entraîner une situation d'urgence pour les membres de l'équipe impactant ainsi l'ensemble du bon déroulement du

projet. Le DDD met en avant la phase d'analyse préalable du projet et son importance, comme le souligne l'intervenant 1 :

"Je pense qu'on a eu trois mois d'avance, plus ou moins, parce qu'il y a eu toute la mise en place technique qui allait permettre de soutenir le DDD justement avec l'écriture d'architecture logique, mais aussi des décisions d'implémentation technique. Le DDD au début de projet a peut-être un petit peu ralenti l'équipe parce qu'il y a beaucoup de règles, beaucoup de mise en place à effectuer."

Pour l'équipe, ce temps nécessaire en début de projet impacte directement la qualité des analyses même si ça n'en augmente pas directement la vélocité :

"Ça m'a aidé à faire de meilleures analyses. On va dire pas spécialement des analyses plus rapides parce que je ne pense pas que le DDD permet d'aller plus vite. Je pense qu'il permet de mieux faire les choses." (intervenant 1)

Celui-ci met également en avant la complexité de la gestion des évènement asynchrones au sein de l'application et du temps nécessaire à leurs développements. Même s'ils s'avèrent être une plus-value au final, leur gestion nécessitent un temps d'analyse et de développement supplémentaire afin d'éviter plusieurs problèmes, comme expliqué dans le point 2.3.4 et souligné par l'intervenant 1 :

"Le temps de développement clairement, c'est la gestion asynchrone des choses parce qu'il faut faire beaucoup plus attention à ce qu'on fait en asynchrone qu'en synchrone. Le fait de devoir gérer certaines modifications de façon asynchrone ralentit quand même l'écriture de l'analyse parce que c'est plus complexe et peut ralentir aussi l'écriture du développement."

Enfin, cette complexité peut être atténuée en fonction de l'expérience du membre de l'équipe. Certains principes du DDD se rapprochent des bonnes pratiques mises en place dans le production de code[100] ou encore dans les méthodes agiles. Dès lors la compréhension de ces principes se voit facilitée, comme expliqué par l'intervenant 1 :

"Mais ce qu'il y a en fait, c'est que les analystes arrivaient à un certain degré d'expérience. Je pense qu'on va, comment dire ça ? Ils ont des concepts d'analyse en tête qui, sans nécessairement le savoir, se rapprochent à mon avis des concepts qu'ils ont expliqués dans le DDD."

6.4.3 Développeurs

L'avis est similaire chez les développeurs qui estiment gagner en temps de développement sur les tests, la régression ou encore les futures évolutions de l'application par rapport au temps de préparation nécessaire en amont. Cette particularité est illustrée par l'intervenant 2 :

"Donc oui les performances ne sont probablement pas à la hauteur lors des premières semaines, mais nous y gagnons lors des phases de régressions, de tests ou encore de modifications. "

Néanmoins, et pour contrebalancer les propos tenus plus haut sur l'expérience, il arrive que certains développeurs éprouvent des difficultés à s'adapter à la nouvelle approche et ses différents Pattern quand ceux-ci possèdent déjà une expérience probante sur d'autres projets, comme le souligne l'intervenant 1 :

"Ce n'est pas toujours évident, notamment chez des développeurs qui ont toujours fonctionné d'une certaine façon. Et tout le monde n'a pas l'esprit ouvert à des nouvelles données, à des nouveaux patterns. "

6.4.4 Défis et obstacles

La mise en œuvre du DDD présente de nombreux défis et obstacles qui peuvent être regroupés en plusieurs catégories. Les interviews et la littérature scientifique mettent en évidence plusieurs de ces défis, notamment la courbe d'apprentissage, la coordination et la communication entre les membres de l'équipe, et la nécessité d'adapter l'approche aux besoins du projet.

L'un des principaux défis est la courbe d'apprentissage associée à la mise en œuvre du DDD. Cette approche nécessite une compréhension approfondie des concepts et des pratiques associées[68]. Les intervenants soulignent la difficulté de maîtriser ces concepts, en particulier pour les développeurs qui ne sont pas experts dans le domaine. Certains mentionnent que le manque de compréhension des concepts du DDD, tels que les Value Objects et les Entities, peut entraîner des problèmes lors de la mise en œuvre de l'approche.

Un autre défi important est la coordination et la communication entre les membres de l'équipe. La mise en œuvre réussie du DDD nécessite une communication étroite et continue entre les experts du domaine et les développeurs[101]. Les intervenants soulignent que cette communication peut être difficile à maintenir, en particulier dans des environnements où les experts du domaine et les développeurs sont géographiquement éloignés ou ont des horaires différents.

Enfin, il est crucial d'adapter le DDD aux besoins spécifiques du projet. Le DDD n'est pas nécessairement approprié pour tous les projets, et il est important de déterminer si l'approche est adaptée aux besoins spécifiques du projet[71]. Les intervenants mentionnent que la mise en œuvre du DDD est plus justifiée pour des projets complexes et de grande envergure. La littérature scientifique souligne également l'importance de sélectionner l'approche appropriée en fonction des caractéristiques et des exigences du projet.

6.4.5 Influence sur les analystes

L'impact du DDD sur les analystes est multiple, allant de l'amélioration de la communication entre les parties prenantes jusqu'à la facilitation de la modélisation du domaine. Les interviews et les recherches scientifiques montrent que l'adoption du DDD peut modifier la façon dont les analystes travaillent et interagissent avec les autres membres de l'équipe.

Tout d'abord, le DDD encourage une communication plus efficace entre les analystes et les autres parties prenantes du projet, comme les développeurs et les experts du domaine[102]. Le langage omniprésent, un concept clé du DDD, permet aux analystes de mieux comprendre et de décrire les besoins du domaine en termes techniques et métier. Cela facilite la collaboration et la communication entre les différents membres de l'équipe, ce qui peut mener à une meilleure compréhension des problèmes à résoudre et des solutions à mettre en œuvre[103].

Ensuite, le DDD favorise une modélisation du domaine plus efficace, ce qui est essentiel pour les analystes. La modélisation du domaine permet aux analystes de mieux comprendre les Entities, les relations et les comportements du domaine, facilitant ainsi la conception de solutions plus adaptées aux besoins des utilisateurs[102]. En adoptant le DDD, les analystes peuvent identifier et aborder les problèmes complexes et les exigences non fonctionnelles de manière plus systématique, ce qui peut conduire à une meilleure qualité des logiciels[104].

Enfin, le DDD peut aider les analystes à mieux anticiper et gérer les changements dans le domaine. Les concepts du DDD, tels que les contextes limités et les Aggregate, permettent aux analystes de décomposer le domaine en unités plus petites et plus gérables, facilitant ainsi l'adaptation aux changements[105]. Les analystes peuvent ainsi concevoir des systèmes plus

évolutifs et flexibles, capables de répondre aux besoins changeants des utilisateurs et du marché[102].

6.4.6 Impact sur les développeurs

L'effet du DDD sur les développeurs est notable, car il influence la façon dont ils conçoivent, développent et entretiennent les applications. Les entretiens et les études scientifiques soulignent que la mise en œuvre du DDD peut améliorer la qualité du code, faciliter la collaboration et aider les développeurs à gérer plus efficacement la complexité du domaine.

D'abord, le DDD peut contribuer à l'amélioration de la qualité du code en incitant les développeurs à élaborer des logiciels modulaires et bien structurés[72]. Les concepts clés du DDD, tels que les contextes bornés et les Aggregate, aident les développeurs à diviser le domaine en parties plus petites et cohérentes. Cela permet de réduire la complexité du code et d'améliorer sa lisibilité, facilitant ainsi la maintenance et l'évolution du logiciel[105].

Ensuite, le DDD facilite la collaboration entre les développeurs et les autres parties prenantes du projet, comme les analystes et les experts du domaine. Le langage ubiquitaire, un élément central du DDD, favorise une compréhension partagée des concepts du domaine, ce qui facilite la communication et la coordination entre les membres de l'équipe[102]. Cette meilleure communication peut conduire à une meilleure compréhension des besoins et des problèmes à résoudre, ce qui peut se traduire par des solutions plus efficaces et adaptées aux besoins des utilisateurs[103].

Enfin, le DDD aide les développeurs à gérer plus efficacement la complexité du domaine en leur fournissant des techniques et des outils pour modéliser et organiser le domaine de manière plus systématique[102]. En adoptant le DDD, les développeurs peuvent mieux comprendre les Entities, les relations et les comportements du domaine, ce qui leur permet de concevoir des solutions plus adaptées et robustes face aux changements[104].

Globalement, le DDD a un impact significatif sur les développeurs en améliorant la qualité du code, en facilitant la collaboration et en aidant à gérer plus efficacement la complexité du domaine. La mise en œuvre du DDD peut ainsi contribuer à une meilleure qualité des logiciels et à une meilleure satisfaction des utilisateurs.

L'adoption du DDD dans des projets complexes présente des avantages et des défis importants, comme l'ont souligné les entretiens et la littérature. Malgré la complexité inhérente à cette approche, les bénéfices à long terme surpassent les obstacles initiaux.

L'un des avantages majeurs de l'adoption du DDD est la facilitation de l'évolution des modèles mentaux partagés, ce qui conduit à une meilleure compréhension du domaine métier et une collaboration

6.5 L'impact des méthodes agiles

L'approche conjointe du Domain-Driven Design (DDD) et des méthodes agiles se concentre sur la collaboration étroite entre les membres de l'équipe de développement et les parties prenantes du projet, visant à produire des logiciels de haute qualité qui répondent aux besoins des utilisateurs. Nos entretiens ont été structurés pour recueillir des informations sur les pratiques de travail de l'équipe, leur attitude face au changement (Q10, Q11, Q12 et Q13), leurs objectifs et résultats (Q21, Q22 et Q23) ainsi que les performances de l'équipe (Q24 et Q25).

Ces approches intégrées, comme l'a souligné l'intervenant 2, facilitent la collaboration au sein des équipes en alignant les Bounded Contexts du DDD avec les Epic des méthodes agiles, et

les agrégats du DDD avec les User Stories. L'intégration réussie du DDD et des méthodes agiles offre des avantages en termes de planification, de conception et de développement, permettant aux équipes de tirer parti des avantages de chaque méthodologie.

Le DDD offre des techniques pour modéliser et organiser efficacement un domaine complexe, tandis que les méthodes agiles mettent l'accent sur la collaboration, l'itération et la flexibilité, favorisant une réponse rapide aux changements des besoins des clients[73]. Ensemble, ces approches permettent aux équipes de concevoir des solutions logicielles robustes et évolutives tout en restant agiles et réactives.

L'intégration du DDD et des méthodes agiles facilite également la communication et la collaboration entre les parties prenantes du projet, y compris les développeurs, les analystes, les experts du domaine et les clients. Le langage omniprésent du DDD et les pratiques de collaboration agile, comme les stand-ups quotidiens et les revues de sprint, aident à forger une compréhension commune des objectifs du projet et des problèmes à résoudre, permettant une meilleure coordination et une prise de décision plus rapide[73].

De plus, selon les témoignages recueillis lors des entretiens, l'adoption du DDD dans les équipes agiles permet de surmonter certaines difficultés couramment rencontrées, telles que les problèmes de bugs ou de mauvaise interprétation des analyses. Le DDD, grâce à sa phase d'analyse approfondie en amont, permet de minimiser ces problèmes, conduisant à une meilleure qualité logicielle et à une réduction des coûts et des délais de développement[69].

En résumé, l'approche conjointe du DDD et des méthodes agiles, en alignant les artefacts de chaque méthode, favorise une collaboration plus efficace au sein des équipes de développement. Cette intégration réduit les difficultés couramment rencontrées par les équipes agiles sans DDD et contribue à une amélioration de la qualité logicielle et à une réduction des coûts et des délais de développement.

6.5.1 Généralités

De l'avis général il semble assez facile d'aligner la structure des artefacts agiles tels les Epic et les User Story avec les découpes présentes dans le DDD comme les Bounded Context ou avec plus de granularité les Aggregate et les Entities, comme expliqué par l'intervenant 2 :

"Plus précisément comme nous utilisons la méthode agile nous avons aligné chaque Bounded Context sur les Epic du projet. De plus chaque aggregat représente une User Story, voir plusieurs si les analystes estiment que la découpe n'est pas assez granulaire. Puis quand la story est définitivement finalisée, celle-ci est encore découpée en plusieurs chapitres."

Certains soulignent également les difficultés que peuvent rencontrer les membres d'une équipe agile n'utilisant pas le DDD. En effet, pour rappel, les méthodes agiles (Scrum) fonctionne par itération successive, souvent faite d'allers-retours entre analystes, développeurs ou testeurs. Le DDD, grâce à sa plus grande préparation en amont du projet, diminue les interactions entre les membres de l'équipe liées aux bugs ou à la mauvaise interprétation des analyses, comme le suggère l'intervenant 4 :

"Il nous a fallu plusieurs mois pour arriver à cerner le métier et commencer un semblant d'analyse. Je pense d'ailleurs que ça a fait un peu peur aux patrons au début, ça prenait trop de temps. Mais maintenant, on est content d'être passé par là, car ça nous évite pas mal de soucis rencontrés habituellement dans les projets agiles. Il y a beaucoup moins d'allers-retours, moins de bugs détectés, moins de questions de la part des développeurs."

6.5.2 Analystes

L'un des aspects fondamentaux des méthodes agiles consiste en l'interaction régulière avec les experts métiers comme expliqué dans le point 2.2. En impliquant les experts métiers dès le début du projet, les analystes peuvent travailler en étroite collaboration avec eux pour comprendre leur langage et leur perspective. Cela permet de créer un modèle de domaine précis qui répond aux besoins réels des utilisateurs. On peut reprendre les explications de l'intervenant 4 à ce sujet reprises dans le point 6.1.2 traitant de l'Ubiquitous Langage et des interactions avec les analystes.

Ces itérations avec les experts métiers sont le plus souvent rythmées par des Events Storming. Même si c'est technique n'est pas directement issue des méthodes agiles (elle a été créée par Albert Brandolini[76]) elle est régulièrement citée dans les articles traitant de l'application des méthodes agiles[106, 107] comme une amélioration significative des interactions avec le métier. Ces Event Storming sont justement utilisés dans le carde de notre sujet, comme le précise l'intervenant 4 :

"Nous avons aussi organisé des Events Storming avec les huissiers afin de capter les grandes idées du métier et pouvoir déjà procéder à une pré-découpe dans l'analyse."

6.5.3 Impact sur les analystes

"L'impact de l'approche conjointe DDD et agile sur les analystes est significatif. Cette approche a été reconnue pour améliorer la communication et la compréhension entre les analystes et les autres membres de l'équipe, notamment les développeurs et les experts métiers. Selon les interviews, il est évident que l'intégration du DDD dans le processus agile facilite le travail des analystes en leur permettant de mieux comprendre les besoins des utilisateurs et de concevoir des solutions plus adaptées.

Un des aspects clés de cette approche est l'utilisation de l'Ubiquitous Language qui favorise une meilleure compréhension entre les analystes et les experts métiers. L'Ubiquitous Language est un langage commun qui est utilisé pour décrire les concepts du domaine, les règles métier et les processus, permettant aux analystes et aux experts métiers de communiquer de manière plus efficace[31]. Cette amélioration de la communication permet aux analystes de mieux comprendre les besoins des utilisateurs et de créer des modèles de domaine plus précis qui répondent aux besoins réels des utilisateurs.

De plus, l'utilisation de techniques telles que l'Event Storming facilite la collaboration entre les analystes et les experts métiers. L'Event Storming est une technique de modélisation qui encourage la discussion et la découverte des concepts du domaine, des événements, des commandes et des Aggregate[76]. Les analystes peuvent ainsi mieux comprendre les processus métier et identifier les zones où des améliorations peuvent être apportées. Cette technique est également utile pour identifier les limites contextuelles (Bounded Contexts) et les Aggregate, éléments clés du DDD.

Enfin, le DDD et agile encourage la collaboration étroite entre les analystes, les développeurs et les experts métiers tout au long du projet. Cette collaboration permet aux analystes de recevoir des retours d'information rapides et de s'adapter en conséquence, améliorant ainsi la qualité des solutions proposées[73]. En outre, cette collaboration étroite facilite la résolution des problèmes et des obstacles potentiels, réduisant ainsi les risques associés au projet et contribuant à la réussite globale du projet[69].

Dans l'ensemble, l'adoption de l'approche conjointe DDD et agile a un impact positif sur les

analystes en améliorant la communication, la compréhension et la collaboration avec les autres membres de l'équipe. Cette approche contribue ainsi à la réussite globale du projet.

6.6 Synthèse et perspectives

En conclusion, l'association du DDD et des méthodes agiles offre de nombreux avantages significatifs. L'Ubiquitous Language favorise une communication claire et cohérente entre les membres de l'équipe et les parties prenantes, ce qui facilite la compréhension des besoins métier et renforce la collaboration entre analystes, développeurs et experts métier.

La structuration du domaine en Bounded Contexts et l'utilisation des Context Maps permettent d'organiser efficacement le projet, facilitant ainsi la création de solutions modulaires et évolutives tout en réduisant la complexité du système. Les Tactical Patterns du DDD contribuent à la compréhension des concepts métier, à l'identification des éléments clés du domaine, et à la conception de solutions répondant aux besoins spécifiques des utilisateurs, facilitant ainsi la maintenance et l'évolutivité du logiciel.

Néanmoins, le DDD présente des défis, tels que la complexité de sa mise en œuvre, la nécessité d'une compréhension approfondie du domaine et la collaboration étroite entre les membres de l'équipe. Ces défis peuvent être surmontés grâce à une formation adéquate, une communication efficace et l'adoption de bonnes pratiques.

La combinaison du DDD et des méthodes agiles permet de tirer parti des forces de chacun, encourageant collaboration, communication et réactivité, tout en garantissant des solutions de haute qualité répondant aux besoins des utilisateurs.

Tout d'abord, cette combinaison renforce les modèles mentaux partagés en facilitant une compréhension commune des concepts du domaine, des processus métier et des besoins des utilisateurs, ce qui améliore la coordination et l'efficacité de l'équipe. Ensuite, grâce à l'adoption conjointe du DDD et de l'Agile, les membres de l'équipe sont mieux alignés sur les objectifs du projet, favorisant ainsi une approche cohérente et collaborative pour la résolution de problèmes et la prise de décision.

De plus, l'utilisation de l'Ubiquitous Language dans le cadre du DDD et de l'Agile facilite une communication claire et cohérente entre les membres de l'équipe, réduisant ainsi les ambiguïtés et les malentendus. Cette communication améliorée permet de mieux partager les modèles mentaux entre les analystes, les développeurs et les experts métier. Enfin, l'association du DDD et de l'Agile encourage l'adaptabilité et l'apprentissage continu au sein de l'équipe. Les membres de l'équipe sont en mesure d'ajuster rapidement leurs modèles mentaux partagés en fonction des retours d'information et des nouvelles informations, ce qui permet d'améliorer continuellement la qualité et la pertinence des solutions proposées.

En résumé, l'effet de l'association du DDD et de l'Agile sur les modèles mentaux partagés est globalement positif, renforçant la compréhension commune, améliorant la communication, favorisant l'alignement des objectifs et encourageant l'adaptabilité et l'apprentissage au sein de l'équipe de développement.

Dans les perspectives futures, plusieurs pistes peuvent être explorées pour enrichir l'association du DDD et des méthodes agiles avec des technologies et approches émergentes.

— Formation et développement des compétences : L'efficacité du DDD et des méthodes agiles dépend en grande partie des compétences et de l'expertise de l'équipe de développement. Il serait bénéfique d'investir davantage dans la formation et le développement professionnel des développeurs, des analystes, des chefs de projet et des autres parties prenantes afin de renforcer leur compréhension et leur maîtrise du DDD et des méthodes agiles.

— Amélioration de la communication et de la collaboration : L'amélioration de la communication et de la collaboration entre les différentes parties prenantes est essentielle pour l'application réussie du DDD et des méthodes agiles. Des outils et des techniques pour faciliter la communication, promouvoir la transparence, résoudre les conflits et encourager la collaboration pourraient être explorés et intégrés dans les pratiques de l'équipe.

— Engagement des utilisateurs et des parties prenantes : L'engagement des utilisateurs et des parties prenantes est crucial pour comprendre les besoins et les attentes, obtenir des retours d'information et assurer l'alignement entre le développement du logiciel et les objectifs du projet. Des stratégies pour promouvoir cet engagement, telles que l'implication des utilisateurs dans le processus de développement, la collecte et l'analyse des retours d'information et la mise en œuvre de changements basés sur ces retours, pourraient être explorées.

— Adoption d'une culture d'amélioration continue : L'adoption d'une culture d'amélioration continue, où l'équipe est encouragée à apprendre de ses erreurs, à expérimenter de nouvelles idées et à s'améliorer constamment, pourrait aider à maximiser les avantages du DDD et des méthodes agiles. Des méthodes pour promouvoir cette culture, telles que les rétrospectives de sprint, le feedback 360 degrés et l'apprentissage organisationnel, pourraient être intégrées.

— Agile Modeling : Cette méthode, qui se concentre sur l'adaptabilité et l'évolution continue, pourrait être intégrée aux pratiques de DDD et Agile pour renforcer l'alignement entre les modèles de domaine et leur mise en œuvre technique. L'Agile Modeling pourrait faciliter la communication au sein des équipes en fournissant des outils visuels pour une meilleure compréhension des objectifs et des exigences du système. De plus, sa flexibilité pourrait contribuer à rendre le processus de développement plus réactif aux changements.

En complément de l'intégration de ces technologies émergentes, il serait intéressant de mener des études de cas et de recueillir des retours d'expérience sur des projets ayant appliqué conjointement le DDD et l'Agile. Ces études pourraient fournir des informations précieuses sur les meilleures pratiques, les enseignements tirés et les défis rencontrés dans ces projets, permettant ainsi d'améliorer continuellement l'adoption et la mise en œuvre de l'approche combinée DDD et Agile.

6.6.1 Limitations de l'étude

L'étude présentée comporte certaines limitations qui doivent être prises en compte lors de l'interprétation des résultats et des conclusions. Tout d'abord, les interviews et les discussions menées pour cette étude proviennent d'un échantillon limité de participants, ce qui peut introduire des biais et limiter la généralisation des résultats à d'autres contextes ou équipes de développement. Il est possible que les expériences et les points de vue des participants ne soient pas représentatifs de l'ensemble de la population concernée par l'association du DDD et de l'Agile.

De plus, l'analyse des résultats repose en grande partie sur les témoignages des participants, qui peuvent être sujets à des erreurs de mémoire, des influences sociales ou le biais de confirmation. Le biais de confirmation se manifeste quand les individus recherchent, interprètent et se souviennent des informations de manière à confirmer leurs idées préconçues, ce qui peut involontairement orienter leurs réponses. Ces facteurs peuvent affecter la validité des conclusions tirées de cette étude. Il serait bénéfique d'inclure des données quantitatives ou des mesures objectives pour renforcer la validité des résultats et mieux évaluer l'impact de l'association du DDD et de l'Agile sur les modèles mentaux partagés.

En outre, les projets et les contextes spécifiques dans lesquels les participants ont appliqué le DDD et l'Agile peuvent influencer les résultats de l'étude. Les défis, les avantages et les expériences rencontrés peuvent varier en fonction de facteurs tels que la taille de l'équipe, la complexité du projet, l'expertise des membres de l'équipe et les contraintes organisationnelles. Il serait donc important d'explorer l'impact de ces facteurs sur les modèles mentaux partagés et de prendre en compte leur influence dans l'analyse des résultats.

Enfin, l'étude se concentre principalement sur l'impact de l'association du DDD et de l'Agile sur les modèles mentaux partagés. Bien que cela représente un aspect important, il serait pertinent d'étudier également l'influence de cette association sur d'autres aspects du développement logiciel, tels que la qualité du code, la productivité de l'équipe ou la satisfaction des utilisateurs, pour avoir une vision plus complète de ses effets.

En tenant compte de ces limitations, il est essentiel de poursuivre les recherches sur ce sujet, en élargissant l'échantillon de participants, en incluant des données quantitatives et en explorant l'influence de divers facteurs contextuels pour renforcer la validité et la généralisation des conclusions.

7 Conclusion

Dans ce mémoire, nous avons abordé les enjeux liés à l'application du DDD pour favoriser les modèles mentaux partagés dans le contexte des projets de développement logiciel, tout en prenant en compte l'intégration avec les méthodes agiles. Les objectifs étaient d'évaluer les avantages et les défis de cette approche et d'analyser l'impact sur la communication, la collaboration et la compréhension entre les membres de l'équipe, ainsi que sur la qualité des solutions proposées.

Dans l'état de l'art, nous avons examiné les concepts clés du DDD, notamment l'importance des modèles mentaux partagés pour une meilleure communication et compréhension des besoins métier. Nous avons également étudié les différentes pratiques et techniques associées au DDD et à l'Agile, ainsi que les synergies potentielles entre les deux approches.

Dans la partie méthodologie, nous avons présenté les différentes étapes de la recherche, y compris la sélection des participants, la réalisation des interviews et l'analyse des données. Les discussions et les résultats issus de ces interviews ont été présentés dans les chapitres suivants, mettant en évidence les bénéfices et les obstacles rencontrés par les équipes lors de l'adoption du DDD pour favoriser les modèles mentaux partagés et en l'associant avec les méthodes agiles.

En conclusion, les résultats de notre étude tendent à montrer que l'application du DDD pour renforcer les modèles mentaux partagés, en combinaison avec les méthodes agiles, offre des avantages considérables en termes de communication, de collaboration et de compréhension entre les membres de l'équipe, ainsi que pour la qualité des solutions proposées. Notre analyse permet de répondre positivement aux deux questions de recherche :

— L'application du DDD dans le développement logiciel influence de manière significative l'évolution des modèles mentaux partagés entre les différents acteurs impliqués dans la conception et la mise en œuvre d'un développement logiciel. En mettant en place les principes du DDD, les équipes sont en mesure de mieux comprendre les exigences métier et d'améliorer la communication et la collaboration entre les parties prenantes.

— L'application conjointe du DDD et des méthodes agiles a également un impact positif sur l'évolution des modèles mentaux partagés entre les différents acteurs impliqués dans la conception et la mise en œuvre d'un développement logiciel. Cette combinaison permet une meilleure adaptation aux changements, une planification plus flexible et une amélioration continue, ce qui renforce encore la compréhension partagée et la collaboration entre les membres de l'équipe.

Les défis liés à l'implémentation du DDD et de l'Agile peuvent être surmontés en mettant en place une formation adéquate, en encourageant une communication efficace et en adoptant les meilleures pratiques.

Ce mémoire souligne l'importance de l'approche DDD dans la promotion des modèles mentaux partagés au sein des équipes de développement logiciel, ainsi que son intégration réussie avec les méthodes agiles. Cette combinaison favorise la réussite des projets en améliorant la communication et la coopération entre les différents acteurs, tout en garantissant des solutions de haute qualité qui répondent aux besoins des utilisateurs. Les perspectives futures pourraient explorer l'investissement dans la formation et le développement des compétences, l'amélioration de la communication et de la collaboration, l'engagement accru des utilisateurs et des parties prenantes, et l'adoption d'une culture d'amélioration continue, ainsi que l'analyse d'études de cas et de retours d'expérience sur des projets ayant appliqué l'approche DDD pour renforcer les modèles mentaux partagés et en l'associant avec l'Agile.

Références

[1] AGILISTE.FR. *Lexique Agile Scrum*. Disponible sur : `https://agiliste.fr/lexique-agile-scrum/`. 2021. URL : `https://agiliste.fr/lexique-agile-scrum/`.

[2] Kai PETERSEN et al. « Systematic mapping studies in software engineering ». In : *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. 2008, p. 1-10.

[3] Maria SALAMA, Rami BAHSOON et Nelly BENCOMO. « Managing Trade-offs in Self-Adaptive Architectures : A Systematic Mapping Study ». In : août 2016, p. 249-297. ISBN : 9780128028551. DOI : `10.1016/B978-0-12-802855-1.00011-3`.

[4] Institut AGILE. *Velocity*. Disponible sur : `http://referentiel.institut-agile.fr/velocity.html`. 2021. URL : `http://referentiel.institut-agile.fr/velocity.html`.

[5] Ken SCHWABER et Mike BEEDLE. *Agile software development with Scrum*. T. 1. Prentice Hall Upper Saddle River, 2002.

[6] Kent BECK. *Extreme programming explained : embrace change*. addison-wesley professional, 2000.

[7] Richard KLIMOSKI et Susan MOHAMMED. « Team mental model : Construct or metaphor ? » In : *Journal of management* 20.2 (1994), p. 403-437.

[8] Catholijn M JONKER, M Birna VAN RIEMSDIJK et Bas VERMEULEN. « Shared mental models : A conceptual analysis ». In : *Coordination, Organizations, Institutions, and Norms in Agent Systems VI : COIN 2010 International Workshops, COIN@ AAMAS 2010, Toronto, Canada, May 2010, COIN@ MALLOW 2010, Lyon, France, August 2010, Revised Selected Papers*. Springer. 2011, p. 132-151.

[9] James M SCHMIDTKE et Anne CUMMINGS. « The effects of virtualness on teamwork behavioral components : The role of shared mental models ». In : *Human Resource Management Review* 27.4 (2017), p. 660-677.

[10] C. Shawn BURKE et al. « Understanding team adaptation : A conceptual analysis and model ». In : *Journal of Applied Psychology* 91.6 (2006), p. 1189.

[11] Cannon E VOLPE et al. « The effects of communication, resource, and task uncertainty on team member's utilization of shared mental models ». In : *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 40.20 (1996), p. 1310-1314.

[12] Beng-Chong LIM et Katherine J KLEIN. « Team mental models and team performance : A field study of the effects of team mental model similarity and accuracy ». In : *Journal of Organizational Behavior* 31.4 (2010), p. 587-605.

[13] Liesje VAN BOVEN et Anita EERLAND. « The shared mental models concept applied to team situational awareness ». In : *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 48.3 (2004), p. 357-361.

[14] John E MATHIEU et al. « The influence of shared mental models on team process and performance. » In : *Journal of applied psychology* 85.2 (2000), p. 273.

[15] Remus Ilies PRITCHARD et Stephanie C PAYNE. « Member familiarity and team development : A retrospective study of the effect of team development on team performance ». In : *Group & Organization Management* 25.2 (2000), p. 224-243.

[16] Michelle A Marks, John E Mathieu et Stephen J Zaccaro. « Temporal issues in team processes ». In : *Academy of Management Review* 26.3 (2001), p. 530-546.

[17] Susan Mohammed, Rebecca S Billings et Jacqueline Barbatis. « The metaphor evaluation approach : A new method for assessing the validity of idiosyncratic semantic interpretations ». In : *The Journal of applied behavioral science* 36.4 (2000), p. 481-503.

[18] Janis A Cannon-Bowers et Eduardo Salas. « Addressing the issue of common method variance : A tutorial ». In : *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. T. 45. 25. SAGE Publications. 2001, p. 1988-1992.

[19] John Rooke, Lauri Koskela et Glenn Howell. « Developing shared mental models in construction projects : An action research study ». In : *Construction Management and Economics* 26.10 (2008), p. 1061-1072.

[20] Kimberly A Smith-Jentsch et al. « Evaluating distributed learning teams using shared mental models ». In : *Advances in Human Performance and Cognitive Engineering Research* 5 (2005), p. 123-161.

[21] Stephen P Borgatti et al. « Network analysis in the social sciences ». In : *Science* 323.5916 (2009), p. 892-895.

[22] Janis A Cannon-Bowers, Eduardo Salas et Sharon A Converse. « Building shared mental models ». In : *Human factors* 35.1 (1993), p. 3-20.

[23] JA Espinosa et al. *The Effect of Shared Mental Models and Knowledge Distribution on Asynchronous Team Coordination and Performance : Empirical Evidence from Management Teams*. Rapp. tech. Working Paper, Carnegie-Mellon University, 2001 (available at http ://www . . ., 2001.

[24] Eduardo Salas, Dana E Sims et C Shawn Burke. « Is there a "big five" in teamwork ? » In : *Small group research* 36.5 (2005), p. 555-599.

[25] Ken Collier. *Agile analytics : A value-driven approach to business intelligence and data warehousing*. Addison-Wesley, 2012.

[26] Véronique Messager. *Gestion de projet agile : avec Scrum, Lean, eXtreme Programming...* Editions Eyrolles, 2013.

[27] Begoña Losada, Maite Urretavizcaya et Isabel Fernández-Castro. « A guide to agile development of interactive software with a "User Objectives"-driven methodology ». In : *Science of Computer Programming* 78.11 (2013), p. 2268-2281.

[28] Kent Beck et al. « Manifeste pour le développement AGILE de logiciel ». In : *En ligne : http ://agilemanifesto. org/iso/fr* (2001).

[29] Ken Schwaber. « Scrum development process ». In : *Business object design and implementation*. Springer, 1997, p. 117-134.

[30] *La méthode Agile comme méthode de travail chez Mercator : Explications*. Accessed : yyyy-mm-dd. Year of Publication. URL : `https://www.mercator.eu/fr/la-methode-agile-comme-methode-de-travail-chez-mercator-explications.chtml`.

[31] Eric Evans et Eric J Evans. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[32] Microsoft. *Tactical DDD patterns in .NET*. `https://learn.microsoft.com/en-us/azure/architecture/microservices/model/tactical-ddd`. Consulté le 25 avril 2023. 2022.

[33] C Bowers et E Salas. « Shared mental models in expert decision making ». In : *Individual and Group Decision Making* 221 (1993).

[34] Susan Mohammed, Lori Ferzandi et Katherine Hamilton. « Metaphor no more : A 15-year review of the team mental model construct ». In : *Journal of management* 36.4 (2010), p. 876-910.

[35] LaToza Thomas, Venolia Gina et Deline Robert. « Maintaining mental models : a study of developer work habits ». In : (2006), p. 1-10. doi : 10.1145/1134285.

[36] Hee-Dong Yang, Hye-Ryun Kang et Robert M Mason. « An exploratory study on meta skills in software development teams : antecedent cooperation skills and personality for shared mental models ». In : *European Journal of Information Systems* 17.1 (2008), p. 47-61.

[37] Melissa A Schilling et Ravi Shankar. *Strategic management of technological innovation*. McGraw-Hill Education, 2019.

[38] Ravindranath Madhavan et Rajiv Grover. « From embedded knowledge to embodied knowledge : New product development as knowledge management ». In : *Journal of marketing* 62.4 (1998), p. 1-12.

[39] Robert R McCrae et Paul T Costa Jr. « Adding Liebe und Arbeit : The full five-factor model and well-being ». In : *Personality and social psychology bulletin* 17.2 (1991), p. 227-232.

[40] Levesque Laurie, Wilson Jeanne et Wholey Douglas. « Cognitive divergence and shared mental models in software development project teams ». In : 22 (2001), p. 135-144. doi : 10.1002/job.87.

[41] Barbara Scozzi et al. « Shared mental models among open source software developers ». In : *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. IEEE. 2008, p. 306-306.

[42] Alberto Espinosa et al. « Shared Mental Models and Coordination in Large-Scale, Distributed Software Development ». In : (2001).

[43] Hala Annabi. *Moving from individual contribution to group learning : The early years of the Apache Web server*. Syracuse University, 2005.

[44] James D Herbsleb et al. « Distance, dependencies, and delay in a global collaboration ». In : *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 2000, p. 319-328.

[45] Alberto Espinosa et al. « Shared Mental Models, Familiarity, and Coordination : A Multi-Method Study of Distributed Software Teams ». In : (2002).

[46] Joan R Rentsch et Richard J Klimoski. « Why do 'great minds' think alike ? : Antecedents of team member schema agreement ». In : *Journal of Organizational Behavior : The International Journal of Industrial, Occupational and Organizational Psychology and Behavior* 22.2 (2001), p. 107-120.

[47] Ric Holt. « Software architecture as a shared mental model ». In : *Proceedings of the ASERC Workhop on Software Architecture, University of Alberta* (2002), p. 64.

[48] David Garlan. « Software architecture ». In : (2008).

[49] Jon R Katzenbach et Douglas K Smith. *The discipline of teams*. Harvard Business Press, 2008.

[50] Tore Dyba. « Improvisation in small software organizations ». In : *IEEE software* 17.5 (2000), p. 82-87.

[51] Sridhar Nerur et VenuGopal Balijepally. « Theoretical reflections on agile development methodologies ». In : *Communications of the ACM* 50.3 (2007), p. 79-83.

[52] Torgeir Dingsøyr et al. *A decade of agile methodologies : Towards explaining agile software development*. 2012.

[53] Paula M Bach et John M Carroll. « Characterizing the dynamics of open user experience design : The cases of Firefox and OpenOffice. org ». In : *Journal of the Association for Information Systems* 11.12 (2010), p. 1.

[54] Jack SC Hsu et al. « Exploring the impact of team mental models on information utilization and project performance in system development ». In : *International Journal of Project Management* 29.1 (2011), p. 1-12.

[55] Arun Rai, Likoebe M Maruping et Viswanath Venkatesh. « Offshore information systems project success : The role of social embeddedness and cultural characteristics ». In : *MIS quarterly* (2009), p. 617-641.

[56] Xihui Zhang et al. « Sources of conflict between developers and testers in software development ». In : *Information & Management* 51.1 (2014), p. 13-26.

[57] Xiaodan Yu et Stacie Petter. « Understanding agile software development practices using shared mental models theory ». In : *Information and Software Technology* 56.8 (2014), p. 911-921. issn : 0950-5849. doi : https://doi.org/10.1016/j.infsof.2014.02.010. url : https://www.sciencedirect.com/science/article/pii/S0950584914000524.

[58] Ann Fruhling et Gert-Jan De Vreede. « Field experiences with eXtreme programming : developing an emergency response system ». In : *Journal of Management Information Systems* 22.4 (2006), p. 39-68.

[59] Renée J Stout et al. « Planning, shared mental models, and coordinated performance : An empirical link is established ». In : *Human factors* 41.1 (1999), p. 61-71.

[60] Karthik Dinakar. « Agile development : overcoming a short-term focus in implementing best practices ». In : *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009, p. 579-588.

[61] Juha Koskela et Pekka Abrahamsson. « On-site customer in an XP project : empirical results from a case study ». In : *European Conference on Software Process Improvement*. Springer. 2004, p. 1-11.

[62] Nils Brede Moe, Torgeir Dingsøyr et Tore Dybå. « A teamwork model for understanding an agile team : A case study of a Scrum project ». In : *Information and software technology* 52.5 (2010), p. 480-491.

[63] Jaak Jurison. « Software project management : the manager's view ». In : *Communications of the association for information Systems* 2.1 (1999), p. 17.

[64] Kude Thomas et al. « How Pair Programming Influences Team Performance : The Role of Backup Behavior, Shared Mental Models, and Task Novelty ». In : (2019). doi : 10.1287/isre.2019.0856.

[65] VenuGopal Balijepally et al. « Are two heads better than one for software development ? The productivity paradox of pair programming ». In : *MIS quarterly* (2009), p. 91-118.

[66] Laurie Williams et al. « Strengthening the case for pair programming ». In : *IEEE software* 17.4 (2000), p. 19-25.

[67] Ikujirō Nonaka et al. *The knowledge-creating company : How Japanese companies create the dynamics of innovation*. T. 105. OUP USA, 1995.

[68] Scott Millett et Nick Tune. *Patterns, Principles, and Practices of Domain-Driven Design*. Wiley, 2006.

[69] T. Widjaja, X. Yuan et C. Teplovs. « The Use of Domain-Driven Design and Agile Methods in the Implementation of ». In : (2016).

[70] D. Haywood. *Domain-Driven Design Distilled*. Addison-Wesley Professional, 2018.

[71] Jimmy Nilsson. *Applying Domain-Driven Design and Patterns : With Examples in C# and .NET*. Addison-Wesley Professional, 2006.

[72] Scott Millett et Nick Tune. *Patterns, Principles, and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[73] E. Hendrickson et S. Bate. *Agile Retrospectives : Making Good Teams Great*. Pragmatic Bookshelf, 2006.

[74] Sébastien Remy et Slimane Zayni. « On the Role of Domain-Specific Languages in Domain-Driven Design ». In : *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2016.

[75] S. Millett et P. Tune. *Patterns, Principles, and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[76] Alberto Brandolini. « Introducing event storming ». In : *blog, Ziobrando's Lair* 18 (2013).

[77] Matthew B Miles et A Michael Huberman. *Analyse des données qualitatives*. De Boeck Supérieur, 2003.

[78] Greg Guest, Arwen Bunce et Laura Johnson. « How many interviews are enough ? An experiment with data saturation and variability ». In : *Field methods* 18.1 (2006), p. 59-82.

[79] Geneviève Imbert. « L'entretien semi-directif : à la frontière de la santé publique et de l'anthropologie ». In : *Recherches en soins infirmiers* 3 (2010), p. 23-34.

[80] Andréanne Gélinas Proulx et Éric Dionne. « Blanchet, A., & Gotman, A.(2007). Série «L'enquête et ses méthodes» : L'entretien (2e éd. refondue). Paris : Armand Colin. » In : *Mesure et évaluation en éducation* 33.2 (2010), p. 127-131.

[81] Luc Van Campenhoudt, Raymond Quivy et Jacques Marquet. « Manuel de recherche en sciences sociales ». In : (2017).

[82] Terry L Dickinson et Robert M McIntyre. « A conceptual framework for teamwork measurement ». In : *Team performance assessment and measurement*. Psychology Press, 1997, p. 31-56.

[83] Emma Nordbäck et Alberto Espinosa. « Cognitive and behavioral leadership coordination : Linking shared leadership to high performance in global teams ». In : *2015 48th Hawaii International Conference on System Sciences*. IEEE. 2015, p. 402-411.

[84] Susan Mohammed et Brad C Dumville. « Team mental models in a team knowledge framework : Expanding theory and measurement across disciplinary boundaries ». In : *Journal of Organizational Behavior : The International Journal of Industrial, Occupational and Organizational Psychology and Behavior* 22.2 (2001), p. 89-106.

[85] Leslie A DeChurch et Jessica R Mesmer-Magnus. « The cognitive underpinnings of effective teamwork : a meta-analysis. » In : *Journal of applied psychology* 95.1 (2010), p. 32.

[86] Finn Olav Björnson et Tone Bratteteig. « The role of a shared understanding of domain-specific languages in software engineering ». In : *Software Process : Improvement and Practice* 13.6 (2008), p. 573-587.

[87] Fonseca Alcides et Ruben Pereira. « The role of bounded context in domain-driven design : A systematic mapping study ». In : *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017.

[88] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013.

[89] Vladimir Milovidov. « Domain-Driven Design in Practice ». In : *2018 IEEE 12th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE. 2018, p. 1-4.

[90] Ioannis Stamelos et al. « Code quality analysis in open source software development ». In : *Information systems journal* 12.1 (2002), p. 43-60.

[91] Bill Curtis, Herb Krasner et Neil Iscoe. « A field study of the software design process for large systems ». In : *Communications of the ACM* 31.11 (1988), p. 1268-1287.

[92] Peter Naur et Brian Randell. *Software engineering : Report of a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, 1976.

[93] Thomas D Latoza, Gina Venolia et Robert DeLine. « Maintaining mental models : a study of developer work habits ». In : *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, p. 492-501.

[94] Jürgen Pichler et João Henrique Pimentel. « Domain knowledge and software development : An empirical investigation ». In : *2013 20th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS)*. IEEE. 2013, p. 3-12.

[95] Torgeir Dingsøyr, Tore Dybå et Nils Brede Moe. *Teamwork quality and project success in software development*. Springer, 2012.

[96] Yair Wand et Ron Weber. « On the deep structure of information systems ». In : *Information Systems Journal* 5.3 (1995), p. 203-223.

[97] Karl Ulrich et Steven D Eppinger. *Product Design and Development*. McGraw-Hill Education, 2015.

[98] David L Parnas. « On the Criteria To Be Used in Decomposing Systems into Modules ». In : *Communications of the ACM* 15.12 (1972), p. 1053-1058.

[99] Sinan Si Alhir. *UML in a Nutshell : A Desktop Quick Reference*. O'Reilly Media, 2003.

[100] Robert C Martin. *Clean Code-Refactoring, Patterns, Testen und Techniken für saubere Code : Deutsche Ausgabe*. MITP-Verlags GmbH & Co. KG, 2013.

[101] Robert Biddle, James Noble et Ewan Tempero. *Communication in Requirements Engineering : A Survey of Australian Software Development Organizations*. IEEE Computer Society, 2006.

[102] Raman Ramsin et Richard F Paige. « Engineering Software : A Comprehensive Guide to Developing High-Quality Software ». In : *IEEE Software* 25.3 (2008).

[103] Rob Wojcik et al. « Documenting Software Architectures : Views and Beyond ». In : *IEEE Software* 23.6 (2006), p. 89-91.

[104] Jon Erickson et Kalle Lyytinen. « The complexity of the software process ». In : *ACM Computing Surveys (CSUR)* 32.3 (2000), p. 177-201.

[105] Len Bass, Paul Clements et Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2012.

[106] Sebastian Copei et al. « From Monolithic Models to Agile Micromodels. » In : *MODELSWARD*. 2022, p. 227-233.

[107] Ömer Uludağ et al. « Supporting large-scale agile development with domain-driven design ». In : *Agile Processes in Software Engineering and Extreme Programming : 19th International Conference, XP 2018, Porto, Portugal, May 21–25, 2018, Proceedings 19*. Springer. 2018, p. 232-247.

8 Annexes