CESI exia
ÉCOLE SUPÉRIEURE D'INFORMATIQUE

Projet de Recherche et Innovation

5ème année

Manager des Systèmes d'Information

Vulgarisation des patrons de conception

Auteur :
Pierrick Knecht

Tuteur encadrant :
Thierry Blanc
Julio Santilario Elena

18 mai 2015

Résumé

Dans ce projet de recherche et d'innovation nous allons aborder le monde des Patrons de Conception, ou plus communément les "Design Patterns". Ce projet vise à répondre aux problématiques suivantes : "Pourquoi les design patterns sont-ils si peu connus/utilisés ?", "Est-il possible de vulgariser le concept de design patterns ?" et finalement "Existe t-il des approches pédagogiques originales pour rendre cet apprentissage plus simple ?". A l'issue de ce projet nous espérons qu'une réponse innovante à ces problématiques émergera, peut être même apparaîtra un premier prototype de pédagogie vulgarisatrice de design pattern.

Abstract

In this research and innovation projet we are going to explore the world of "design patterns". This project will try to bring an answer to the followings questions : "Why the design patterns are not well known/unused ?", "Is it possible to simplify the design patterns ?" and finally "Are there any kind of original learning support to make this learning easier ?". At the end of this projet we will try to find an innovative solution for this problematics maybe even a first prototype of polurizer learning support.

Remerciements

Je voudrais tout d'abord remercier M. Herold & M. Guillotin professeurs d'informatique dans la formation du Brevet de Technicien Supérieur en Informatique et Réseaux pour l'Industrie et les Services, qui m'ont initié à la programmation Orientée Objet et qui m'ont encouragé à poursuivre mes études.

Je remercie également M. Thierry Blanc tuteur pédagogique dans la formation de Manager des Systèmes d'Informations à l'Exia.CESI, pour ses échanges et son suivi continu sur ce projet.

Je remercie M. Christian Gauthier tuteur de stage en 4e année dans l'entreprise de EFS, qui a su stimuler ma capacité à réfléchir et aussi pour ses encouragements et son intérêt manifeste pour le sujet de l'UML et des Design Patterns.

Je souhaiterais remercier M. Ronan Presle étudiant en 5e année dans la formation de Manager des Systèmes d'Informations à l'Exia.CESI, pour les multiples échanges lors de nos phases d'analyses durant les projets et pour les nombreuses questions qu'il est venu m'apporter.

Je remercie également M. Kevin Rouze étudiant en informatique et ami, pour sa disponibilité et son professionnalisme en tant que cobaye durant la conception du prototype nécessaire à ce projet.

Et pour finir je remercie ma famille et mes amis pour leur aide et leur soutien durant tout ce projet.

Table des matières

1 Introduction

Ce Projet de Recherche et Innovation (PRI) est réalisé dans le cadre de ma formation en Manager de Systèmes d'information à l'Exia.Cesi. Ce projet prend donc place dans un contexte purement informatique. L'ensemble des sujets traités dans ce document s'inscrivent uniquement dans ce domaine.

Quelle forme pourrait prendre une solution pédagogique vulgarisatrice qui chercherait à rendre plus accessible le langage Unified Modeling Language (UML) et l'utilisation des Design Patterns (documentation, exemple parlant), le tout en s'appuyant sur les pédagogies actuelles ? Ce projet vise à répondre à cette problématique.

Nous allons dans un premier temps faire un "État de l'art". Celui-ci sera construit autour de quatre notions nécessaires à la compréhension de notre étude.

La première partie de cet état de l'art expliquera le plus simplement possible ce qu'est l'UML, comment il s'est inscrit comme un incontournable du monde de l'informatique et ce qu'il apporte aux différentes phases lors du développement d'une solution logicielle.

La seconde notion sera les Design Patterns. Nous allons chercher à comprendre leur nature, la façon dont ils ont été crées, comment ils sont devenus incontournables pour le métier de l'informatique, et finalement le rôle qu'ils jouent dans une architecture logicielle.

Ensuite nous nous intéresserons aux différents outils de modélisation UML utilisés par la communauté des développeurs. La compréhension de ces outils de modélisation n'est pas critique pour l'étude néanmoins elle demeure importante puisque la quasi-intégralité des schémas proviendront de l'un des ces outils.

Enfin une quatrième et dernière partie sera dédiée aux différents formats d'apprentissage des design patterns utilisées par la communauté des développeurs.

Après cet état de l'art nous allons plonger dans l'étude même de notre PRI. Comme expliqué auparavant ici l'objectif sera de répondre aux différentes hypothèses que nous avons soulevées. Ces hypothèses sont toutes axées autour du sujet des Design Patterns, à savoir : "Les gens rencontrent-ils des difficultés à ap-

prendre le fonctionnement des Design Patterns ?", "Existe-il un manque en terme de solution d'apprentissage sur le sujet des Design patterns ?" et finalement "Quel forme prendrait une solution d'apprentissage vulgarisatrice sur le sujet des design patterns ?".

Cet étude est donc construite en trois temps afin répondre aux mieux aux différentes problématiques posées précédemment.

Cette étude s'appuiera sur une enquête qui vise à récolter des éléments de connaissance sur l'apprentissage des design patterns. Cette enquête permettra de valider ou d'invalider nos différentes hypothèses. L'intérêt de cette méthode est de faire apparaître l'existence d'éventuels problèmes rencontrés lors de cet apprentissage. L'objectif est simplement de comprendre au mieux pourquoi le problème se présente et de savoir si il est possible d'apporter une solution.

Ensuite nous découvrirons la conception d'un prototype d'apprentissage qui doit permettre la vulgarisation d'un sujet que nous pensons actuellement peu accessible. Ce prototype sera mis à disposition d'un étudiant cible, une personne apte à apprendre le sujet des design patterns mais qui y est pour l'instant complètement étranger.

Plus tard nous chercherons un format d'apprentissage différent afin de s'approcher d'un support plus appropriés qui permettra de répondre aux besoins pédagogiques. L'intérêt de se nouveau support est de donner une forme innovante à notre prototype, une forme dans l'ère du temps. Puisque qu'à la fin l'objectif est d'imaginer une forme et un fond à cette solution car nous souhaitons imaginer une solution complète, un "**recueil sur les design patterns**".

Et finalement nous conclurons sur cette thèse afin de prendre du recul sur l'intégralité du travail effectués ces derniers temps.

2 État de l'art

Dans cet état de l'art nous allons découvrir l'UML et l'ensemble des éléments qui le composent. Ensuite nous nous intéresserons au concept des Design Patterns. Plus tard nous nous pencherons sur les différents outils de modélisation UML, des logiciels populaires auprès de la communauté de développeurs. Finalement nous nous pencherons sur les formats pédagogiques existants.

2.1 UML

2.1.1 Définition

"En informatique UML (de l'anglais Unified Modeling Language), ou Langage de modélisation unifié, est un langage de modélisation graphique à base de pictogrammes. Il est utilisé en développement logiciel, et en conception orientée objet. UML est couramment utilisé dans les projets logiciels"[1]

De manière plus concrète l'UML est un langage technique qui a pour objectif de faciliter la communication et l'explication lors du développement d'un logiciel basé sur la programmation orientée objet. Il est aussi supposé offrir une syntaxe aussi sophistiquée et complète qu'une langue réelle tel que le français ou l'anglais. Les mots et les phrases sont substitués par des diagrammes, des mots-clés et des pictogrammes.

On dit souvent qu'une analyse bien faite permet d'assurer la réussite du développement d'une application. Tout d'abord il est important de savoir que l'architecture de l'UML s'appuie sur le modèle d'architecture "vue 4+1" de Philippe Kruchten. Le concept est qu'une architecture peut se découper en différentes perspectives complémentaires entre elles.[1][2]

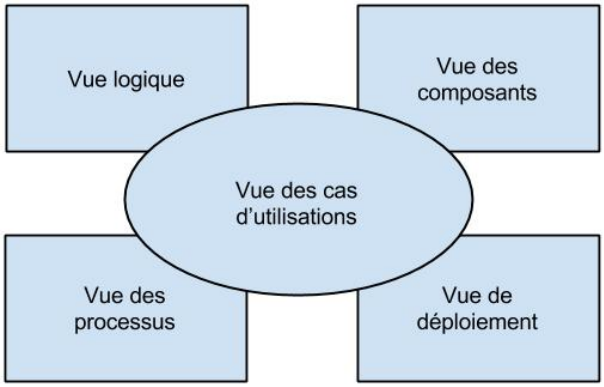

FIGURE 1 – le modèle d'architecture "Vue 4+1"

- La **vue logique** est une vue de haut-niveau, elle représente l'abstraction et l'encapsulation. Elle modélise les principaux éléments et mécanismes du système. Elle permet l'identification des éléments du système, leurs relations et interactions.
- La **vue d'implémentation** est une vue de bas-niveau, elle représente la réalisation du système. Elle montre donc les dépendances entre les composants, les contraintes de développement, et une organisation des modules en sous-systèmes.
- La **vue des processus** est la vue la plus importante dans le cas d'environnements multitâches. Elle permet le découpage du système en tâches, d'évaluer leurs interactions, et de synchroniser les threads.
- La **vue de déploiement** est la vue la plus importante dans le cas d'environnements distribués. Elle décrit la répartition des ressources matérielles et la disposition de ces ressources dans les différents environnements.
- La **vue cas d'utilisations** est la vue permettant de synthétiser le besoin des utilisateurs. Elle est primordiale pour identifier le besoin de chaque acteurs du système. Cette vue est le noyau, elle permet d'unifier les autres vues.[2]

Une analyse à l'aide de l'UML se fait en plusieurs étapes, toutes ne sont pas strictement obligatoires, mais mises bout à bout permettent à une personne extérieur au métier de l'informatique (tel qu'un client, un commercial, ou un extérieur) de comprendre le fonctionnement de l'application.

Pour chaque étapes du développement un ensemble de diagrammes sont utilisables. La solution logicielle dans son intégralité peut donc contenir plus d'une dizaine de diagrammes différents.

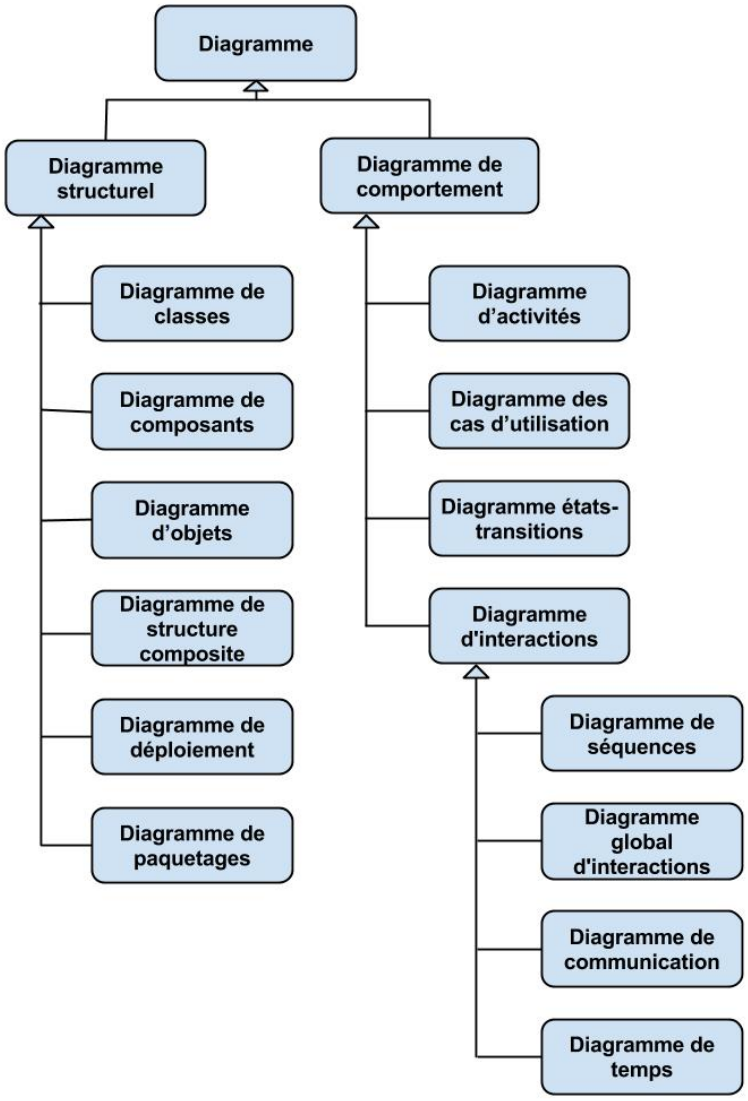

FIGURE 2 – Découpage diagrammes

Ces 13 diagrammes ont été classés en deux familles, les diagrammes structurels et les diagrammes comportementaux (qui contiendront eux-même une sous-famille appelé diagrammes d'interactions).

- Les **diagrammes structurels** permettent d'identifier les différents intervenants internes aux systèmes (composants physiques, éléments matériels, conteneur logiques, ...). Ils permettent donc de modéliser la structure statique du système.

    - Le **diagramme de classe** permet une représentation logique des différentes classes qui constituent le système. Nous reviendrons sur ce diagramme plus tard, car il demeure capital pour le reste de l'étude.
    - Le **diagramme de composants** permet de poser les différents composants systèmes (fichier, librairies, BDD...) nécessaires à la solution.
    - Le **diagramme d'objets** permet de faire apparaître les différentes instances d'une classe utilisée dans le système.
    - Le **diagramme de structure composite** permet de représenter les relations entre différents composants d'une classe.
    - Le **diagramme de déploiement** permet de représenter l'ensemble du matériel nécessaire au fonctionnement et sa disposition dans le système.
    - Le **diagramme de paquets** apporte une représentation organisée du regroupement des paquets, de leur hiérarchisation (conteneurs logiques), et de leurs dépendances.[1][3]

- Les **diagrammes comportementaux** quant à eux permettent de se focaliser sur le comportement du système (sous forme de machines à états ou de flux), et les différentes interactions (l'échange de messages, ou même une variation de données), en somme le comportement dynamique du système.

    - Le **diagramme d'activités** décrit un processus sous la forme d'une série d'activités, il permet donc de représenter un système ou une application de manière séquentielle.
    - Le **diagramme de cas d'utilisation** permet de représenter l'ensemble des interactions entre le système et les acteurs (intervenants extérieurs,

exemple un utilisateur).

- Le **diagramme états-transitions** sert à représenter un composant sous forme de graphes d'états, reliés par des arcs orientés servant à décrire les transitions.
- Le **diagramme d'interactions** permet de décrire les particularités d'une communication précise entre objets.

  + Le **diagramme de séquences** est une représentation chronologique des interactions avec ou au sein du système.
  + Le **diagramme de global d'interactions** permet de décrire les enchaînements possibles entre les scénarii préalablement identifiés sous forme de diagrammes de séquence (variante du diagramme d'activité).
  + Le **diagramme de communication** offre une représentation spatiale des objets et de leurs interactions.
  + Le **diagramme de temps** offre une description des variations d'une données au cours du temps.[1][3]

Pour finir, l'ensemble des diagrammes ont des modèles d'éléments qui leurs sont propres. Ces modèles d'éléments se présentent sous la forme de petits pictogrammes où chacun apporte une notion unique.

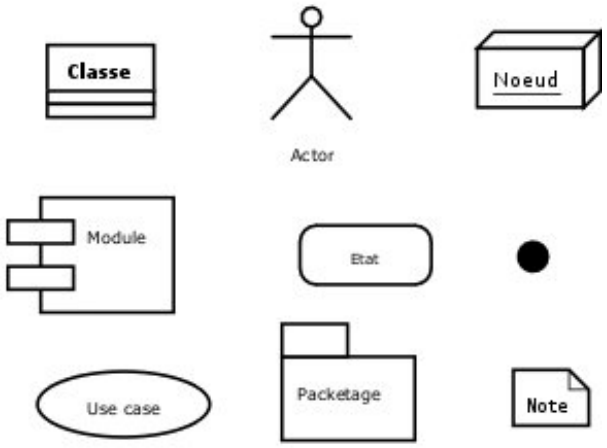

FIGURE 3 – Éléments UML

2.1.2 L'histoire de l'UML

Tout commence dans les années 90, à l'époque un grand nombre de livres traitant sur les logiciels de modélisations orienté-objet (OO) ont été publiés, introduisant ainsi les premières notations graphiques. On parle déjà de la méthode Booch (dont le concepteur est Grady Booch) et OMT (ayant pour créateur James Rumbaugh).

La méthode Booch permet de faciliter l'implémentation d'un programme orienté objet indépendamment du langage dans lequel la solution est étudiée. Il est constitué fondamentalement de classes, d'objets, de méthodes, et d'attributs.

OMT (Technique de Modélisation d'Objet) illustre les relations statiques entre les classes et objets du système. C'est l'ancêtre du diagramme de classes et d'objets.

Cela a donc pris quelques années avant qu'un quelconque standard tente d'apparaître. À la conférence OOPSLA (Object-Oriented Programming, Languages & Applications) de 1995, Grady Booch et Jim Rumbaugh annoncent leur concept *Unified Method*, on parle alors de UML v0.8. Peu de temps après, Rumbaugh & Booch sont rejoint par Ivar Jacobson, on les surnomme alors les "Three Amigos".

Les livres écrit par Booch, Rumbaugh & Jacobson sont très populaires et très appréciés, on dit donc que l'UML est né. Il a fallu attendre jusqu'au 11 Novembre 1997 pour que l'OMG (Object Management Group) - l'association en charge de la standardisation et la promotion du modèle objet - accepte l'UML comme un réel standard. On dit que c'est à ce moment là que la RTF (Revision Task Force) fut créée. [3]

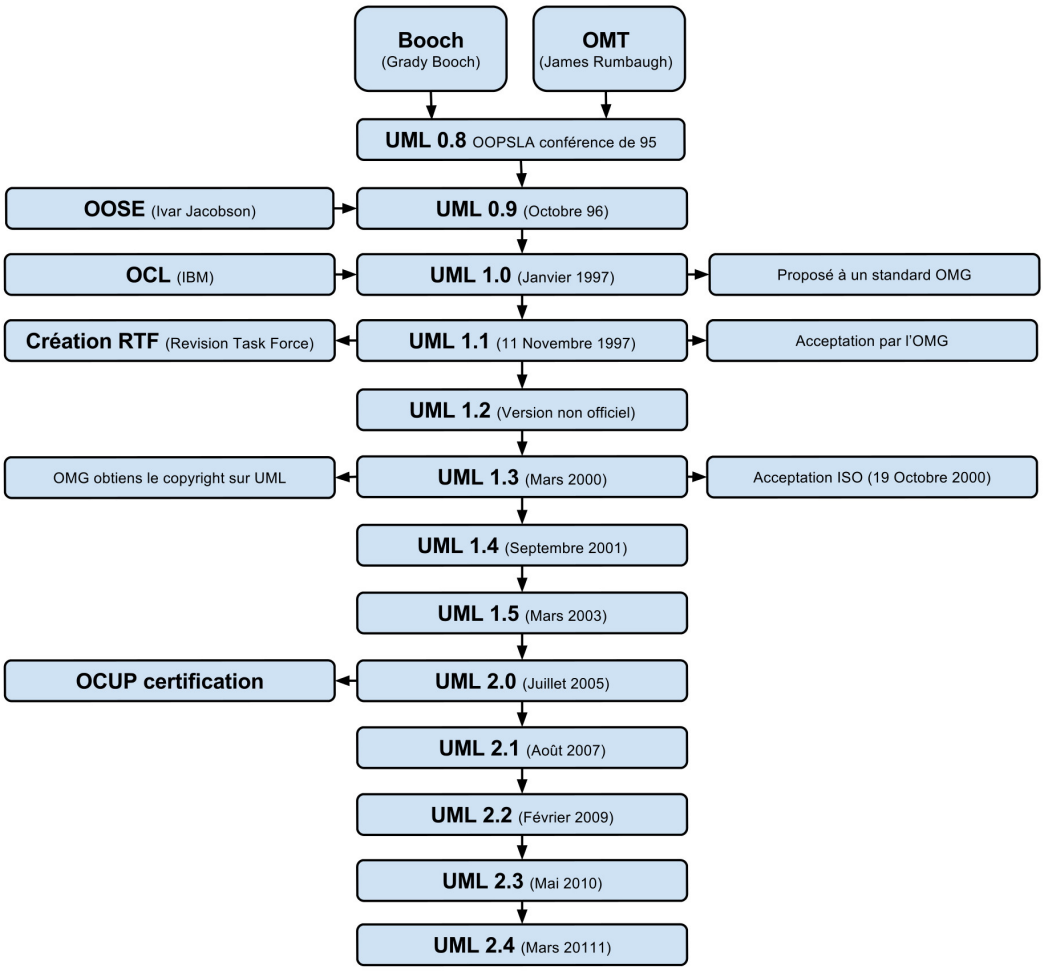

FIGURE 4 – Historique

Au fil des années les différentes évolutions de l'UML se sont succédées :

- Les versions qui ont précédées la **version 1.1** (la version officiellement acceptée par l'OMG) n'étaient en soit que de simples prototypes. On parle à l'époque d'ébauches de la première vrai version de l'UML.
- La **version 1.2** de l'UML n'a jamais été officiellement déployée, elle s'est contenté de n'être qu'une version bêta (elle n'apporter que quelques corrections concernant des erreurs de frappes ou grammaticales).
- La **version 1.3** de l'UML apporte de nombreux changements au niveau des méta-modèles, de la sémantique, et des notations. Ceci reste tout de même une évolution mineure par rapport à la version originale.

- La **version 1.4** de l'UML apporte des améliorations principalement visuelles mais pas nécessairement compatibles avec la version 1.3. Les artefacts y ont été ajoutés pour permettre la représentation physique des composants.
- La **version 1.5** intègre les éléments d'Actions apparaissant dans les diagrammes d'activités et défini le concept de flux de données portés entre deux "Actions".
- La **version 2.0** va quant à elle apporter beaucoup plus d'améliorations. Un ensemble de nouveaux diagrammes (Diagramme d'objets, de paquets, de structures composites, d'interactions, de temps, etc..). Le diagramme d'activités et de séquences a été amélioré. De nombreux stéréotypes ont été retirés (destroy, facade, friend, thread, etc..). Les 13 diagrammes basiques sont divisés en 2 catégories entre les diagrammes de modélisation structurels et les diagrammes de modélisation comportementals.
- La **version 2.1** ne sera qu'une révision mineure de la 2.0, apportant seulement quelques légers correctifs.
- La **version 2.2** amènera juste un peu plus de cohérence à la version 2.1 (2.1.1 et 2.1.2).
- La **version 2.3** sera une révision mineure de la 2.2, clarifiant un peu plus les associations et mettra à jour le diagramme de composant, de structure composite et d'activités.
- La **version 2.4** sera une révision mineure de la 2.3 avec de nouveaux correctifs de bugs, une mise à jour des classes et paquets. Les stéréotypes changent à nouveaux avec l'apparition du stéréotype "Metaclass". [4][5]

2.1.3 La version 2.4 d'UML

L'UML est un sujet intéressant mais surtout très vaste. Sans doute même trop pour être étudié en seulement 2 ans. Nous avons donc eu besoin de nous imposer des limites dans notre recherche. Dans le cadre de notre PRI, nous avons donc décidé de nous arrêter à la version UML 2.4, nous excluons donc toutes évolutions postérieures à cette version.

Nous pouvons nous demander : "Qu'est ce que la version 2.4 ?". Tout d'abord il est important de comprendre l'histoire de l'UML, car même s'il ne semble plus en évolution (sa dernière évolution majeur reste celle de 2005 avec la 2.0) l'UML continue d'avancer. Il est important de retenir que très récemment est sortie la version 2.4.1 qui apporte une révision de la version 2.3 avec quelques correctifs et améliorations des classes et packages. Pour information, une version bêta de la 2.5 est en cours de chantier.[3]

2.1.4 Conclusion sur l'UML

Maintenant que l'UML a été parcouru dans son ensemble, pourquoi avoir choisi d'intégrer l'UML dans notre PRI (Projet de Recherche et Innovation) ?

Arrivé à un certain moment, dans son parcours professionnel l'informaticien n'a plus besoin d'approfondir ses connaissances d'une technologie, mais au contraire il doit plutôt chercher à prendre du recul sur les technologies dans leur ensemble. La force de l'ingénieur logiciel ne réside pas uniquement dans sa maîtrise technique ou dans sa capacité à communiquer, mais dans la qualité de son analyse.

Selon moi l'UML et les Design Patterns s'inscrivent le mieux dans cette objectif de rechercher la qualité. Ce sont tout les deux des outils puissants qui valent la peine d'être appris.

2.2 Les Design Patterns

Les Design Patterns sont au cœur de ce PRI. Durant notre étude nous allons étudier des Design Patterns bien précis, il est donc nécessaire que quelques lignes explique ce concept.

Nous allons donc voir l'origine des patterns, leurs cheminement jusqu'à l'informatique, la spécificité des 23 design patterns crée par les "Gang of Four" et pour finir son application dans les diagrammes de classe en UML.

2.2.1 L'histoire des Patterns

Christopher Alexander est depuis toujours considéré comme le père du "Pattern Language". Créateur de 253 patterns qui dans leur ensemble représente le "Pattern Language", il est aussi à l'origine de l'apparition des Patterns dans l'informatique. Les Patterns étaient à l'origine destinés à être utilisés dans le domaine de l'anthropologie et de l'histoire de l'art. Par la suite ils ont été réutilisés dans le design, et finalement en informatique. Il faudra attendre les années 90 pour que le concept exploré par Christopher Alexander suscite l'intérêt. Les Design Patterns apparaissent par la suite lors des phases de conception de logiciels.

Comme expliqué précédemment le "Pattern Language" était destinés à l'origine au design & à l'architecture. Par exemple pour l'architecture ces 253 patterns sont donc triés dans le "Pattern Language" en 3 grandes familles : "Ville", "Batiment", "Construction".

Il existe donc 253 Patterns connus qui ne sont pas spécialement prévus dans le cadre d'un développement informatique, au vu de leur nombre il était nécessaire de sélectionner quelques design patterns sur lesquels notre étude se porterait. Nous avons donc choisi de nous limiter aux 23 Design Patterns issus du livre de 1995, présentés dans cette étude comme les GOFs (design pattern créer par les Gang of Four). [6]

2.2.2 Les 23 Design Patterns des "Gangs of Four"

Le 21 octobre 1994, Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (connus aussi sous le nom des "Gang of Four") publie le livre "Design Patterns : Elements of Reusable Object-Oriented Software". Le livre présente les 23 Design Patterns considérés comme des références dans le monde de l'Informatique. [7]

Définition : *Un Design Pattern (en français un patron de conception) est un modèle schématique permettant d'apporter une solution à un problème d'architecture logiciel. Les Design Patterns sont nés au début des années 70 des travaux de l'architecte Christopher Alexander.*

Les Design Patterns peuvent être divisés en trois grandes familles :

- Les modèles de **création** :
  Ces patrons sont constitués d'éléments qui échangent avec les mécanismes de création, tentant de créer un objet d'une manière spécifique à une situation.
- Les modèles de **structuration** :
  Ces patrons sont chargés d'alléger une structure en identifiant la façon la plus simple de réaliser la communication entre différentes entités.
- Les modèles de **comportement** :
  Ces patrons sont chargés d'améliorer la flexibilité d'exécution des communications.

2.2.3 Les diagramme de classe et les design patterns

Dans le domaine de l'informatique les design patterns s'utilisent dans les diagrammes de classes. Nous allons approfondir le diagramme de classes, les différents modèles d'éléments et les notations.

Un diagramme de classe se compose donc d'éléments appelés classe. Ils sont représentés sous la forme de rectangles segmentés en 3 parties :

- Leur noms. Le nom de la classe qui sera finalement utilisé dans le code lors de la phase de développement.
- Leurs attributs. Les différents paramètres dont est constitué la classe. Ces informations peuvent être publics, privés, ou protégés.

- Leurs méthodes. Un ensemble d'instructions utilisable au sein de la classe ou même parfois depuis l'extérieur.

Ces classes ou "rectangles" sont reliés par des relations qui peuvent prendre la forme de différentes flèches. Ils existent alors 4 types de relations différentes :

- L'association. Elle représente la relation la plus faible entre deux classes. Celle-ci représente simplement le fait qu'une classes peut en utiliser une autre. La relation peut être uni ou bidirectionnelle.

- L'agrégation. Elle représente une relation un peu plus complexe entre deux classes. L'élément qui a pour agrégat un autre élément, n'a pas un besoin strict du second objet.
  Exemple : un livre n'a pas besoin d'une reliure pour être un livre. Elle peut ne pas être présente. C'est une agrégation.

- La composition. Elle symbolise une relation encore plus forte que l'agrégation. Celle-ci représente le fait qu'une classe doit être constituée d'une autre.
  Exemple : Un livre ne peut se passer de pages. Sans ces pages le livre n'a pas de raisons d'exister. Il en est composé.

- L'héritage. Lorsqu'une classe hérite d'une autre, elle récupère l'ensemble de ses caractéristiques. Elle est la relation la plus forte des quatre.
  Exemple : L'ADN, si un ensemble de filles hérite des gènes de la mère, en informatique une modification de l'ADN de la mère modifierait automatiquement les gènes de ses filles.

Les design patterns vont donc prendre la forme de schéma (cf. Figure 5), constitués en grande partie de classes et joue sur les différents niveaux de relations

(association, agrégation, composition, héritage). Ils vont pouvoir être réemployés dans tous les types de projets à partir du moment où ils répondent à un besoin bien spécifique. Un besoin auquel ils sont supposés répondre. [5][8]

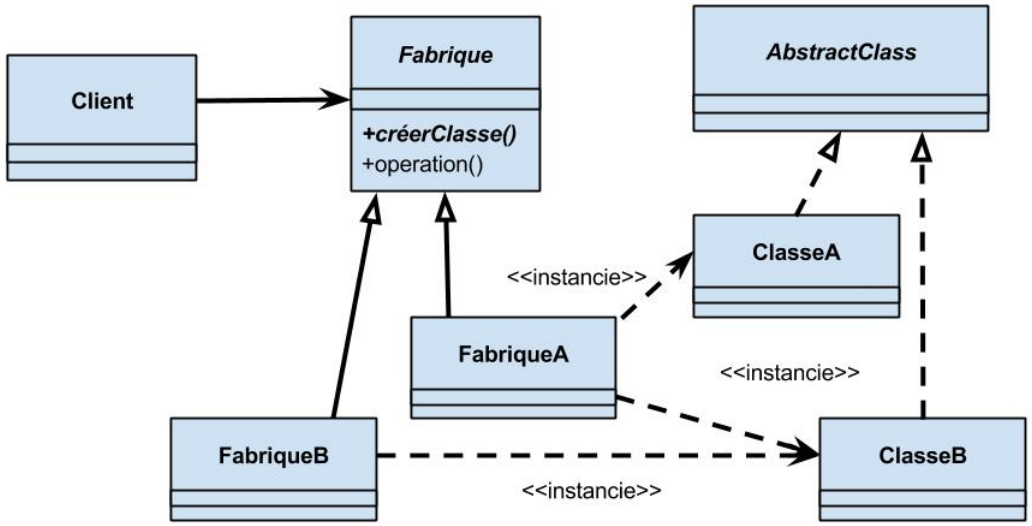

FIGURE 5 – exemple de **design pattern**

2.3 Les outils de modélisation UML

Définition : Un outil de modélisation UML est un logiciel qui permet la réalisation de diagrammes UML, permettant généralement la conception d'une application dans l'ensemble de ses étapes.

Elle demeure un très bon support tout du long de l'analyse. Cependant tous les outils de modélisation UML sont différents, certains intègrent la génération du code sous différents langages, là où une autre va offrir une modélisation des diagrammes plus simplifiés. Ce qui a permis l'apparition d'une grande variété d'outils de modélisation UML.

Dans le cadre de notre étude nous aurions à utiliser certainement un outil de modélisation UML. Il est important de connaître au moins un outil de modélisation UML. C'est l'outil principalement utilisé durant les phases d'analyses basés sur l'UML et sur les design patterns. À défaut de pouvoir déjà tous les présenter, nous avons orienté notre choix sur 2 grands classiques en plus d'un choix personnel. Et pour nous faire une idée de ceux qui étaient populaires nous avons récupéré un sondage basé sur un échantillon de 402 personnes.[9]

2.3.1 Rational Rose

Rational Rose est une solution de développement basée sur des modèles pour le développement de systèmes complexes. Rational Rose est un logiciel propriétaire, il a été vendu pour 2,1 milliard de dollars à IBM, c'est donc une solution payante. Un outil de modélisation UML orienté-objet qui permet de couvrir l'ensemble des phases du cycle de vie de la conception d'un logiciel. Il offre aux analystes des outils pour l'analyse des problèmes, la définition des systèmes et la gestion des besoins. En plus il permet le développement basé sur des modèles pour automatiser le développement, et la génération du code entièrement exécutable.

2.3.2 ArgoUML

ArgoUML est un AGL orienté UML écrit en Java. L'intérêt de cet AGL est d'accompagner l'utilisateur lors de phases d'analyse pour la conception d'un logiciel en le guidant à travers les différentes notations UML. ArgoUML est sous la licence EPL 1.0, le code source d'ArgoUML étant disponible sur internet, il est possible de participer au développement et à l'évolution du logiciel. ArgoUML supporte sept

diagrammes différents : le cas d'utilisation, classe, séquence, état, collaboration, activité et déploiement.

2.3.3 WhiteStarUML

WhiteStarUML est un outil de modélisation UML, issue d'un Fork de StarUML, il apporte une continuité moderne à l'ancienne version en mettant à jour le code avec une édition plus récente de Delphi. StarUML et donc WhiteStarUML permettent la gestion de diagrammes spécifiés dans la norme UML 2.0. WhiteStarUML est sous une licence modifiée de GNU GPL. Il permet l'export des diagrammes sous format JPEG et WMF.

2.4 Approches pédagogiques

Depuis plusieurs années les supports utilisables pour dispenser des connaissances se sont multipliés :

- Les **livres**,
- Les **sites spécialisés**,
- Les **vidéo hébergées** en ligne.

2.4.1 Les livres

On peut trouver aujourd'hui de très nombreux livres traitant sur les sujets de l'UML ou des design patterns. Peu traitent des deux en même temps, et encore moins nombreux sont ceux cherchant à accessibiliser ces connaissances. Ces livres sont plus ou moins spécifiques, ils sont parfois généralisés, et parfois spécialisés.

Le nombre de livres traitant sur le sujet de l'UML s'élève à plusieurs centaines (691 livres différents) sur le site de vente en ligne nommé amazon.fr (site dit spécialisé en vente de livre en ligne). Ce nombre impressionnant de livres est largement justifié par l'évolution fréquente de l'UML ces dernières années.

Quant aux livres traitant sur le sujet des design patterns ils sont encore plus nombreux puisqu'on peut en trouver plus d'un millier (1521 livres différents) toujours sur le même site. Ce nombre est explicable puisque comme dit plus tôt, les design patterns se retrouvent dans d'autres domaines que celui de l'informatique.

Il existe une famille de livres vulgarisateurs connue sous le nom de "pour les nuls" ou "for dummies" en anglais. Ces livres sont dédiés à la vulgarisation de sujets divers et variés. Néanmoins malgré mes recherches aucun livre traitant de l'UML ou des design patterns n'ont été édités[10].

2.4.2 Les sites spécialisés

Depuis plusieurs années de plus en plus de sites apparaissent pour proposer un apprentissage en ligne. Nombreux sont les sites penchés sur le domaine de l'informatique et tous ont leur section dédiée à l'UML et parfois même au design patterns.

Plusieurs sites très célèbres pour leurs cours en ligne proposent quelques chapitres concernant le sujet de l'UML et des design patterns sans couvrir correctement le sujet. Chacun propose une initiation à la notion d'objet, puis explique le

fonctionnement d'un design pattern.[11] L'approche est simple mais elle se contente simplement de recopier des livres existants. Quand aux sites spécialisés sur le sujet des design patterns, leur approche est généralement très technique, et s'appuie essentiellement sur du code (que ce soit en C++, en Java, ou en C♯).[12] Cependant ils constituent de bonne source de connaissance dès lors qu'on se montre capable de comprendre l'anglais.

2.4.3 Les vidéo en ligne

Ce dernier format est encore le plus intéressant. Les idées sont neuves, et le format proposé est généralement très ingénieux. On permet ainsi de synthétiser rapidement un grand nombre d'idées.

Le défaut de ce dernier format est généralement le manque de maturité de ce support. La vidéo en ligne est un format très populaire ces dernières années. On peut donc facilement trouver des cours de langages informatiques destinés aux néophytes. Néanmoins peu de personnes proposent des contenus de qualité sur des sujets plus techniques.

Généralement les vidéos en ligne ne sont pas très riches en images. Nous pouvons des fois y apercevoir quelques impressions d'écrans issues d'un environnement de développement. D'autres fois ce sont juste des diapositives sur lesquels ont entend une voix enregistrée.

La vidéo en ligne est un support intéressant mais qui nécessite plus de travail en apportant par exemple des schémas ou des dessins.

2.5 Conclusion

Dans ce chapitre nous avons donc vu la nature et le rôle de l'UML, nous avons compris les événements qui ont amenés l'UML à ce qu'il est aujourd'hui, l'ensemble des éléments qui le composaient (les vues, les diagrammes, les modèles d'éléments), et pour finir nous avons identifiés la limite de notre étude sur le sujet.

Nous nous sommes penchés sur les design patterns, l'histoire qui a guidé le concept jusqu'au domaine de l'informatique, les 23 design patterns crées par les GOFs. Nous avons pris le soin de revenir sur le diagramme de classe afin de comprendre comment se construisait schématiquement un design pattern.

Ensuite nous avons parcouru les différents outils de modélisation UML utilisables pour la schématisation de design patterns.

Et finalement nous avons listé les différents formats pédagogiques utilisables pour notre recueil.

Durant notre étude nous allons tenter de réfléchir à une solution. Une nouvelle sorte de recueil qui permettrait à tout à chacun de s'appuyer sur les design patterns dans le cadre d'un projet informatique.

3 Étude

Cette section a pour but de présenter l'ensemble des méthodes mises en place pour répondre à la problématique présentée en début de ce document.

Tout d'abord nous allons commencer par voir comment nous avons validé et/ou invalidé nos hypothèses. Ceci permettra d'écarter l'expérience personnelle, et donc de baser notre étude sur une vision plus globale de l'apprentissage des design patterns.

Plus tard, nous présenterons nos résultats, ceux concernant le prototype de recueil de design patterns. Le prototype a pour but d'offrir un approche pédagogique complète, sans qu'elle soit ne nécessairement originale.

Un dernier chapitre sera consacré à la forme que prendre cette solution. Pour rappel ce format devra permettre la mise en place la plus adapté à notre prototype pédagogique.

3.1 Enquête

On observe fréquemment des difficultés qu'ont certaines personnes lorsqu'ils tentent de comprendre le fonctionnement des design patterns. Lorsque nous parlons aujourd'hui de design pattern, les échos sont souvent négatifs sur le sujet. Nombreuses sont les personnes qui ont conservé une mauvaise expérience sur le sujet. Et encore plus nombreuses sont celles qui sont sorties de cette expérience sans avoir maîtrisé la connaissance attendue.

Ceci est donc notre première problématique : "Les développeurs rencontrent-ils réellement des difficultés à utiliser les design patterns ?"

N'ayant pas été capable de trouver des informations répondant à nos hypothèses, il nous a été nécessaire de créer un support qui permettrait enfin d'apporter les réponses nécessaire à cette thèse. Nous avons donc mis en place un questionnaire. Ce questionnaire a donc été soumis à un échantillon précis de personnes représentatifs des utilisateurs de design patterns : les étudiants et les développeurs expérimentés. Les jeunes étudiants sont ceux qui ont été le plus fréquemment exposé à l'apprentissage des Design Patterns. Les développeurs expérimentés sont ceux qui sont le plus à même d'avoir à utiliser les design patterns.

Dans le cas de la première catégorie nous avons fait le choix de soumettre cette

enquête aux différents étudiants de l'école Exia.CESI situé sur le centre de Lyon. Des personnes avec qui nous avons eu l'occasion d'échanger des points de vue à propos de cette problématique, de récolter leur opinions et qui ont été à la source de notre questionnement sur ces difficultés d'apprentissage. Effectivement c'est auprès de cet échantillon que nous sommes arrivés à nous demander pourquoi les développeurs rencontraient autant de difficultés à apprendre.

La seconde catégorie de personnes est constituée des différentes personnes contribuant à la communauté du site developpez.com[9]. Cette communauté est à notre sens l'une des plus actives et des plus représentatives des développeurs tournés vers le partage de connaissance sur internet. La communauté a facilement accepté de répondre à nos questions, ce qui nous a permis de baser notre enquête sur un échantillon de plus de **70 personnes**.

Les chiffres issus de cette enquête sont révélateurs. Comme le montre le sondage sobrement nommé "Enquete 1" (situé en annexe A.2), il est évident que l'apprentissage des design patterns n'est pas aussi accessible que ce nous pourrions penser. Il est important de constater que près de **45% des personnes interrogées rencontrent des difficultés lors de l'apprentissage** des design patterns. Ce qui représente presque une personne sur deux, sachant que plus de **6% de personnes interrogées** préféreront des méthodes plus simples à aborder. Ceci ne prouve pas que ce sujet est plus difficile qu'un autre mais qu'il existe une certaine inaccessibilité dans cet apprentissage. Ceci apporte des éléments de réponse à ma première problématique : "Existe-il aujourd'hui des difficultés à apprendre le fonctionnement des Design Patterns ?".

La seconde question que nous nous sommes posée est : "Existe-il un manque en terme de solution d'apprentissage des design patterns ?". Nous pouvons voir que sur ce même échantillon de personne l'enquête (cf annexe A.3, "Enquete 2") a révélé que **57% des personnes considèrent que "oui il existe un manque"**, que un peu moins de 25% de personnes considèrent que "non" et un peu plus de 18% n'ont tout simplement pas souhaité répondre. Si on ne s'intéresse qu'aux voix exprimées l'enquête élève l'**opinion positive à plus de 69%**.

La dernière hypothèse est la plus critique puisqu'elle a pour but de révéler l'importance des design patterns dans le travail des développeurs. L'enquête a proposé une petite mise en situation de la manière suivante : "Vous faites face à un nouveau collègue qui vous explique que vous pouvez solutionner votre problème

par la mise en place d'une Abstract Factory". Nous leur avons donc posé la question si elles seraient enclin à la mise en place de ce design pattern ou si elles pencheraient plus vers la mise en place d'une solution plus "personnelle". Encore une fois les résultats sont probants (cf Annexe A.4, "Enquête 3"), plus de **81% des personnes sont disposées à mettre en place un design pattern** et donc d'inclure la notion de design pattern dans leur environnement de travail, s'il s'avère bénéfique pour eux. Ceci prouve donc que les design patterns ont une place à gagner dans le monde de l'entreprise auprès de ceux qui ne l'utilisent pas déjà au quotidien.

Bien entendu cette enquête inclut d'autres réponses que nous n'utiliserons pas pour le moment. Puisque l'utilité d'une nouvelle solution est prouvée nous allons à présent voir les différents résultats de ce travail.

3.2 Les résultats

Après de nombreuses recherches, nous avons fait le constat qu'il manquait une dimension à l'apprentissage des design patterns. Précédemment nous avons démontré qu'il existait un besoin pédagogique, et que les développeurs étaient d'accord sur le fait qu'il fallait apporter un plus à la formule d'apprentissage actuelle.

Pour l'instant l'apprentissage des design patterns se fait en deux parties. La première partie comme expliqué plus tôt se base principalement sur des schémas. Ces schémas sont souvent un agencement d'entités constituant le design pattern, puisque pour rappel le design pattern se veut être un patron de conception. Nous parlons de patron de conception dans ce cas, puisqu'il s'agit d'un agencement préconçu qu'il suffit d'insérer dans la solution logicielle existante. Ces schémas sont donc généralement sous la forme de diagrammes que l'on retrouve principalement dans l'UML. Ces diagrammes sont appelés "Diagrammes de Classe". Ces diagrammes se veulent être une représentation visuelle d'une combinaison de classes et de relations. La seconde partie se présente comme une démonstration technique. Cette démonstration se constitue d'un ensemble de segments de code nécessaires à la mise en place du design pattern. Ces extraits de code est directement utilisable, puisqu'il suffirait simplement de le recopier au sein du code de notre projet. Bien entendu quelques adaptations seraient nécessaires pour correspondre correctement aux besoins de la solution logicielle.

Actuellement ces deux possibilités d'apprentissage se sont révélées efficaces. Néanmoins la solution que nous présentons dans cette thèse devrait apporter une troisième facette à la solution d'apprentissage actuelle. Ce qui permettra, en plus de compléter la solution actuelle, d'accessibiliser cette connaissance qui peut apporter un réel plus-value au monde de l'entreprise.

3.2.1 Le prototype

La vulgarisation sous toutes ses formes peut être faite à partir du moment où la personne qui fait le choix d'instruire maîtrise le sujet dont elle parle, se soucis de la clarté de son propos et s'assure d'éviter au maximum l'utilisation de termes techniques qui nécessite un passé sur la connaissance.

Pour présenter ma méthode de vulgarisation nous avons fait le choix de trois design patterns afin de tester mon prototype. Trois design patterns dont nous connaissons le fonctionnement, que nous avons déjà eu l'occasion de mettre en place, et qui imageront assez facilement la vulgarisation des patterns.

Notre choix c'est donc tourné vers les trois design patterns suivants :

- L'**adaptateur**
- Le **décorateur**
- Le **singleton**

En plus de présenter une vulgarisation de ces trois design patterns, nous appliquerons à chaque fois les deux autres méthodes d'apprentissage des design patterns. Ce qui signifie une présentation par la **schématisation** et la **codification**. Pour construire le pattern sous forme de code nous avons fait le choix d'utiliser le Java puisqu'il nous est plus familier. Néanmoins d'autres langages peuvent tout à fait s'adapter à la démonstration tel que le C♯ ou le C++. Nous reviendrons sur le choix de la langue dans le format de la solution finale.

3.2.2 L'adaptateur

L'adaptateur fait partie de la famille des "*Structural design patterns*". Il joue donc de la façon dont peuvent s'imbriquer classe et objet pour obtenir de nouvelles fonctionnalités. [13]

Définition : *"L'adaptateur encapsule les accès à une interface qui ne correspond pas à vos normes ou à vos besoins."*[9]

A - Vulgarisation

L'adaptateur peut être assez facilement assimilé à des éléments de la vie quotidienne. Pour faire simple, un adaptateur (ou *adapter* en anglais) est un élément intermédiaire qui va permettre à un élément A de s'imbriquer dans un élément B alors qu'ils ne peuvent pas en temps normal.

L'élément A souhaite agir sur l'élément B alors qu'ils ne peuvent pas interagir l'un sur l'autre par une différence de caractéristiques. En termes simples si nous devions comparer ça à un objet de la vie courante ce serait l'équivalent d'une poignée, ce qui permettra donc à l'élément A d'influer sur l'élément B.

Sans poignée l'interaction avec le mécanisme de la porte est impossible, il est nécessaire d'avoir une pièce qui s'intègre bien dans le mécanisme de la porte et qui offre une bonne prise en main. Ceci permettra donc à n'importe qui d'utiliser le mécanisme de la porte sans avoir la nécessité de mettre les doigts dans les mécanismes.

En d'autres termes l'adaptateur est un intermédiaire qui permet à deux éléments d'interagir entre eux alors qu'ils ne le pourraient pas en temps normal.

B - Schématisation

A présent comment mettre en place ce design pattern ? Pour ce faire il est important de comprendre comment reproduire la situation. Pour commencer simplement, nous allons tout d'abord reprendre l'explication précédente. Une fois recrée dans notre outil de modélisation, voici le résultat :

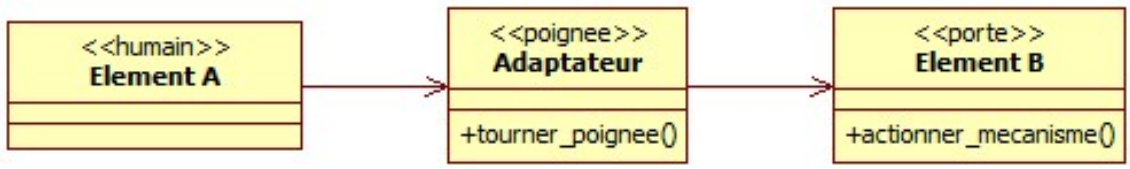

FIGURE 6 – Adaptateur vulgarisé

Nous avons donc une personne (l'élément A) qui va tourner la poignée (la méthode "tourner poignee"), ce qui demandera donc à la poignée d'agir sur le mécanisme de la porte (la méthode "actionner mécanisme").

Bien entendu il ne s'agit pas simplement d'un adaptateur par action. Un adaptateur peut avoir plusieurs méthodes qui peuvent faire agir l'élément B différemment. Il suffira à l'élément A de demander l'action 1 (tourner poignée par exemple), l'action 2 ou l'action 3 si elles ont été crées préalablement. Ceci demandera donc à l'adaptateur d'agir sur l'élément B de la façon 1, de la façon 2 ou de la façon 3.

C - Codification

Maintenant comment allons-nous coder ceci ? Simplement en s'appuyant sur notre schématisation.

Pour mettre en place votre design pattern il faudra effectuer un certain nombre d'étapes.

- Identifiez la classe **Élément_A**. Bien entendu le nom "élément A" est générique, elle prendra le nom qu'il faudra pour votre projet. Cette classe sera votre point de départ, la classe que vous souhaiterez utiliser dans votre application.
- Créez la classe **Adapter**.
- Identifiez la classe **Élément_B**. Tout comme **Élément_A** le nom ici est générique. Cette classe représentera votre finalité.

- Créez une instance de votre classe **Adapter** dans votre **Élément_A**.
- Créez une instance de votre classe **Élément_B** dans votre **Adapter**.
- Créez une méthode **"to adapt"** dans votre class **Adapter** qui se chargera d'appeler la méthode de **Élément_B**.

Bien entendu si nous ne retrouvons pas d'élément A ou B c'est que nous sommes dans un cas où nous ne rencontrons pas le besoin d'un adaptateur. Nous mettons en place un adaptateur uniquement si nous sommes confrontés à deux éléments qui doivent travailler ensemble mais qui n'en sont pas capables. La démonstration de la classe A et B ne sert que de contexte d'utilisation. Nous pouvons très bien y intégrer d'autres choses. L'important ici est de retrouver l'ensemble des éléments nécessaires au fonctionnement de l'adaptateur.

Ci dessous quelques segments de code démonstratifs. Ils servent d'exemple de code et ne doivent pas être considérés comme une façon absolue de coder un design pattern.

```
public class Element_A {
    private Adapter a;

    public Element_A(){
        a = new Adapter();
    }

    public void save_text_Element_B(String s){
        a.to_adapt(s);
    }
}
```

FIGURE 7 – classe Element_A

On peut voir dans la figure 7 que la classe **Element_A** souhaite que **Element_B** conserve des données sous forme de chiffre entier or l'**Element_A** n'a que du texte à lui transmettre. Il l'envoie donc à son instance de la classe **Adapter** puisque **Element_B** ne sait pas travailler avec des informations sous forme de texte (String)

```java
public class Adapter {
    private Element_B eb;

    public Adapter(){
        eb = new Element_B();
    }

    public void to_adapt(String s){
        int i = Integer.parseInt(s);
        eb.save_data(i);
    }
}
```

FIGURE 8 – classe adapter

Dans la figure 8 la classe **Adapter** joue le rôle d'un traducteur. Ici il traduit le texte sous forme de chiffre. Une fois le message transformé il le transmet à "**Element_B**".

```java
public class Element_B {
    int data;
    public Element_B(){
        data = 0;
    }

    public void save_data(int i){
        data = i;
    }
}
```

FIGURE 9 – classe Element_B

La figure 9 montre que **Element_B** travaille avec l'information sous forme de chiffre d'entier comme convenu.

3.2.3 Le décorateur

Le décorateur est lui aussi de la famille des "*Structural design patterns*". Il joue donc de la façon dont peuvent s'imbriquer classe et objet pour obtenir de nouvelles fonctionnalités. [13]

Definition : *"Le décorateur ajoute des fonctionnalité à un objet de manière entièrement dynamique. Parfois plus efficace que l'héritage, il permet d'étendre les capacités de plusieurs classes sans en modifier les sources."*[9]

A - Vulgarisation

Un décorateur est quelque chose de très commun dans notre quotidien. Un décorateur est un concept que l'on retrouve couramment dans la vente de voitures. Par exemple nous nous rendons chez notre concessionnaire pour acheter une voiture X. Le concessionnaire nous propose alors quelques options supplémentaires comme des vitres électriques, un plus grand coffre, ou même la radio Bluetooth.

Ici le besoin initial est la voiture, les options sont donc les petits plus apportés par le concessionnaire. Nous avons donc trois éléments :

- Votre besoin initial (la voiture générique) ;
- Vos options supplémentaires (besoins optionnels) ;
- Votre achat final (qui sera donc la somme des deux).

En entrant dans la boutique nous ne connaissions pas les options qui pourraient être possibles sur la voiture, elles sont apportées par le concessionnaire de manière dynamique (puisque ce n'était pas initialement prévu). Néanmoins vous saviez que vous vouliez une voiture qui puisse rouler, et qui ait un moteur thermique, des besoins définitifs, nous dirons même statiques. Le concessionnaire ici jouera le rôle de décorateur. Il offrira donc le même modèle que notre voiture générique (notre besoin initial) avec des fonctionnalités supplémentaires.

Le décorateur permet donc d'apporter des fonctionnalités optionnelles dynamiquement autour d'un élément générique qui ne nécessite pas de changer son fonctionnement premier.

B - Schématisation

A présent comment mettre en place ce design pattern ?

Pour ce faire il est important de comprendre comment reproduire la situation. Reprenons d'abord l'explication précédente. Une fois recrée dans notre outil de modélisation, voici le résultat :

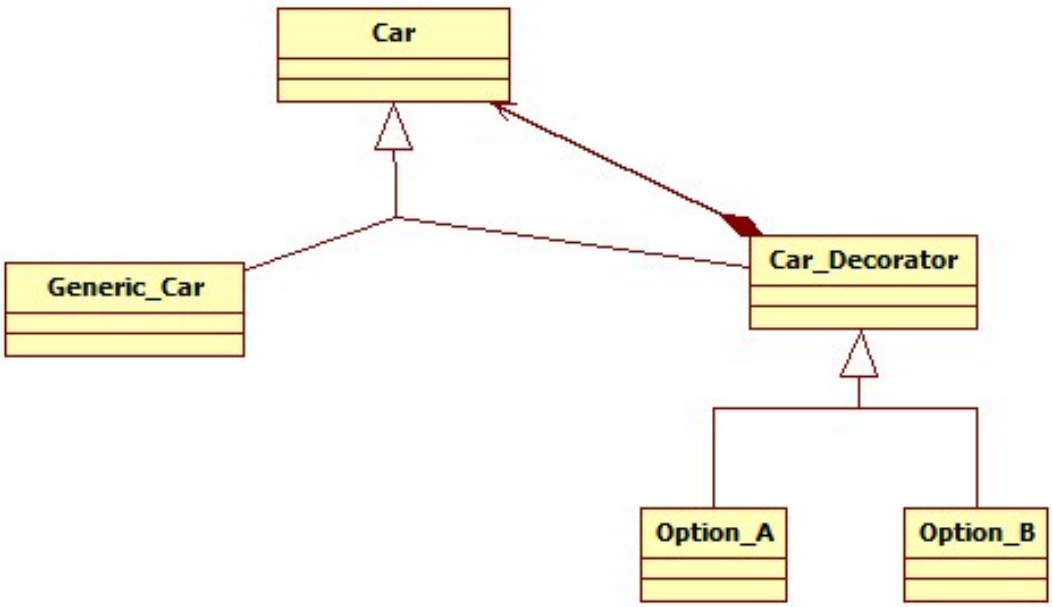

FIGURE 10 – Décorateur vulgarisé

Nous retrouvons donc la voiture souhaitée sous le nom de Generic_Car (qui représente notre besoin initial), le décorateur sous le nom de Car_Decorator (qui représente la possibilité d'une voiture personnalisable) et les différentes options sous les noms de Options_A ou B (qui représentent les vitres électriques, le grand coffre, etc.).

C - Codification

Maintenant nous allons voir comment coder ceci en s'appuyant simplement sur notre schématisation.

Pour mettre en place ce design pattern il faudra effectuer un certain nombre d'étapes.

- Identifiez la classe que vous souhaitez dynamiser. Dans notre cas elle s'appellera **Car**. L'idéal dans le cas présent serait d'utiliser une *"interface"* puisqu'elle ne sera jamais utilisée directement. Nous utiliserons plutôt ces classes filles ;
- Héritez deux classes filles de l'interface **Car**. Une classe **Generic_car** comportera le fonctionnement générique et une classe **Car_Decorator** permettra la personnalisation de la classe **Car** ;
- Héritez autant de fois la classe **Car_Decorator** que vous souhaitez créez d'options à votre classe **Car** ;
- Créez une variable de type **Car** (on parle ici de "*Composition*") dans votre classe **Car_Decorator**. Ce qui permettra à l'option utilisée de contenir elle même une option, qui contiendra elle même une option, qui contiendra elle même... etc.

Ci dessous quelques segments de code démonstratifs. Ils servent d'exemple de code et ne doivent pas être considérés comme une façon absolue de coder un design pattern.

```
public interface Car {
    public List<Car_Decorator> list_option = new ArrayList<>();

    public void drive();
}
```

FIGURE 11 – classe Car

On peut voir dans la figure 11 que la classe **Car** est en fait une interface. Ce qui signifie qu'on ne peut pas avoir d'instance de la classe **Car** et donc forcer l'utilisation des classes filles. Ici on y ajoute une méthode appelée **Drive**, qui représentera le fonctionnement standard attendu de la voiture souhaitée.

```java
public class Generic_car implements Car{

    public Generic_car(){

    }

    @Override
    public void drive() {
        System.out.println("Ma voiture roule.");
    }

}
```

FIGURE 12 – classe Generic_Car

Dans la figure 12 la classe **Generic_Car** hérite donc de sa classe mère **Car**. On y définit le fonctionnement standard de la classe **Car**.

```java
public class Car_Decorator implements Car{

    protected Car voiture;

    public Car_Decorator(Car v){
        this.voiture = v;
    }

    @Override
    public void drive() {
        this.voiture.drive();
    }

}
```

FIGURE 13 – classe Car_Decorator

La figure 13 montre comment la classe **Car_Decorator** conserve l'information de la classe passé en paramètre au constructeur **Car_Decorator()**.

Cette information est capitale pour provoquer le phénomène de cascade car dans le cas du décorateur nous allons fonctionner sur un système de couches empilées. La première couche sera le **Generic_Car**, sur laquelle nous allons superposer une **Option_A**, sur laquelle nous allons superposer une **Option_B**. Lorsque nous appellerons notre objet empilé, chaque couche appellera la couche en dessous d'elle jusqu'à l'appel de la couche la plus basse (le fonctionnement générique).

Ainsi avec un seul objet on permet d'utiliser un fonctionnement personnalisé en fonction du choix des couches empilées. La seule chose qui reste inchangé est la couche la plus basse. Dans notre cas la classe **Generic_Car**.

```
public class Option_A extends Car_Decorator{
    public Option_A(Car v){
        super(v);
    }

    @Override
    public void drive(){
        System.out.println("Ma voiture à l'option A !");
        super.drive();
    }
}
```

FIGURE 14 – classe Generic_Car

Dans la figure 14 on voit un exemple d'option. Que ce soit Option_A ou B la structure reste la même seul le contenu de la méthode **Drive** changera. Le plus important ici est d'hériter de **Car_Decorator**.

```
public static void main(String[] args) {
    Car voiture = new Generic_car();
    voiture.drive();
    // Texte : Ma voiture roule.

    voiture = new Option_A(voiture);
    voiture.drive();
    // Texte : J'ai l'option A!
    //         Ma voiture roule.

    voiture = new Option_B(voiture);
    voiture.drive();
    // Texte : J'ail 'option B!
    //         J'ai l'option A!
    //         Ma voiture roule.
}
```

FIGURE 15 – classe Generic_Car

Logiquement une fois cet ensemble de code mis en place il suffit d'utiliser correctement cet ensemble de classes pour déclencher son fonctionnement en cascade. La figure 15 montre comment utiliser ce design pattern.

3.2.4 Le singleton

Le singleton est de la famille des "Creational design patterns". Cette famille de design pattern joue principalement sur les mécanismes de création d'objets pour permettre des manipulations d'objets situationnelles. [1]

Definition : *"Le singleton s'assure de l'unicité d'une instance de classe et évidemment du moyen d'accéder à cette instance unique."*[9]

A - Vulgarisation

Si un singleton pouvait prendre la forme d'un objet courant dans la vie, ce serait celui d'une boîte aux lettres. Une boîte aux lettres accessible par tous et visible de n'importe où, comme si nous avions posé une boîte aux lettres au milieu d'une rue et qui serait commune à tout le monde.

Imaginons que nous voulions transmettre un message à notre voisine d'en face sans pouvoir directement l'atteindre. Nous nous rendons à cette boîte aux lettres, nous y laissons un message à l'intérieur puis nous repartons. Notre voisine peut venir à tout moment à la boîte aux lettres récupérer notre message sans même avoir besoin de nous connaître. Nous sommes sûrs qu'il s'agit bien de notre lettre puisque nous l'avons laissé dans une boîte aux lettres précise. Aucun intermédiaire n'a pu mettre la main dessus et apporter une modification.

L'intérêt du singleton est donc d'apporter une boîte aux lettres qui permet d'assurer l'unicité de l'information et son accès depuis n'importe où dans le programme.

B - Schématisation

Dans la programmation orienté objet, une problématique ressort souvent. Cette problématique est simplement celle de l'instance unique, ou celle de la donnée unique.

Lorsque vous créez une instance d'une classe vous obtenez un objet. Nous n'avons plus la maîtrise sur la durée de vie de cette objet et ce dès lors que l'on sort du corps d'une méthode. De plus si nous passons cette instance en paramètre d'une méthode, nous ne faisons en soit que passer une copie de cet objet. Nous ne travaillons pas directement sur l'objet original, nous ne maîtrisons donc pas l'unicité de cet objet.

Comment travailler sur un objet unique depuis n'importe où dans notre programme en étant sûr qu'il n'est pas déjà détruit ? Le singleton va apporter cette réponse à l'aide d'un mécanisme des plus ingénieux. Pour ce faire il est important de comprendre comment reproduire la situation.

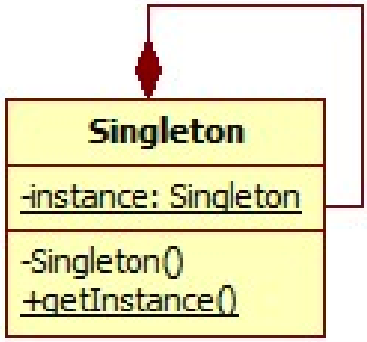

FIGURE 16 – Singleton vulgarisé

Dans la figure 16 nous pouvons voir la forme que va prendre ce design pattern. Ce qui est intéressant de voir ici c'est déjà qu'il n'est constitué que d'une classe.

Tout d'abord il est intéressant de remarquer que l'on ne peut pas instancier cette classe puisque comme on peut le voir le constructeur est privé. Ensuite on peut constater que la classe **Singleton** contient une variable nommée "**instance**" qui est de type **Singleton**. Et pour finir nous pouvons voir apparaître un accesseur nommé "**getInstance**" qui permet d'accéder à l'attribut "**instance**".

Un autre point intéressant, la variable instance et la méthode **getInstance()** sont statiques (représentées par le surlignage de leur noms). La méthode **getIns-**

tance() est *statique* pour permettre d'être utilisable sans nécessité d'instance de la classe **Singleton**, nous n'avons donc pas besoin d'objet de type Singleton.

Nous pouvons donc directement utiliser la méthode **Singleton.getInstance()** sans autre manipulation. La méthode **getInstance()** retourne donc l'attribut "instance", la variable nécessite alors d'être statique puisqu'elle est manipulée par une méthode elle-même statique.

L'agencement de ces éléments semble très complexe mais ils vont permettre la mise en place cette boîte aux lettres.

C - Codification

```
public class Singleton {

    static private Singleton instance;
    private String message;

    private Singleton() {}

    static public Singleton getInstance() {
        if (Singleton.instance == null)
            Singleton.instance = new Singleton();
        return Singleton.instance;
    }

    public void setMessage(String s) {
        this.message = "\n\""+s+"\"";
    }

    public String getMessage() {
        return this.message;
    }
}
```

FIGURE 17 – classe Singleton

Dans la figure 17 on peut voir comment se définit la méthode **getInstance()**. Tout d'abord nous testons si dans la classe **Singleton** l'attribut instance existe, si il n'est pas "*null*". Si c'est le cas nous l'instancions grâce au constructeur privé et donc accessible uniquement depuis l'intérieur de la classe. Ensuite nous renvoyons en valeur de retour l'attribut instance, puisque à l'origine c'est ce que nous voulons

que notre accesseur fasse.

Ensuite nous rajoutons un nouvel attribut que nous allons appeler message. A l'origine notre singleton joue le rôle d'une boîte aux lettres, mais nous pourrions aussi y ajouter un entier, un nombre flottant, un caractère ou même un objet.

Nous avons fait le choix d'ajouter un attribut, nous mettons donc en place un accesseur et un mutateur puisque nous voulons récupérer notre message ou en mettre un autre.

```
public class Me {

    public void putIntoMailBox(String s) {
        Singleton.getInstance().setMessage(s);
    }
}
```

FIGURE 18 – classe Me

Dans la figure 18 nous voyons comment mettre un message dans la boîte aux lettres. Nous appelons la méthode statique de la classe **Singleton** par l'extrait de code **Singleton.getInstance()**, ce qui nous permet d'accéder à l'instance de la classe **Singleton**, elle même statique. Dans cette instance il existe un attribut "*Message*" (cf figure 17) dans lequel nous pouvons laisser un message à l'aide de notre mutateur setMessage.

```
public class Neighbor {

    public String openMailBox() {
        return Singleton.getInstance().getMessage();
    }
}
```

FIGURE 19 – classe Neigbhor

Dans la figure 19 nous voyons comment le voisin(e) récupère notre message grâce à l'accesseur **getMessage()** de l'attribut "**instance**".

3.3 Le cobaye

Nous avons pu voir l'application de notre prototype de vulgarisation sur trois design patterns différents.

Cette approche a été exposée à un analyste programmeur et étudiant en informatique au niveau Bac+3. Cette personne était à l'origine entièrement étrangère au concept mais a accepté l'expérience de ce nouveau support pédagogique. Grâce à l'appui de cette personne nous avons pu améliorer notre prototype jusqu'à un niveau d'apprentissage satisfaisant sur le sujet.

Nous avons conçu une première version du prototype avec un premier design pattern que nous lui avons soumis. Cette personne nous a fait un retour sur les difficultés rencontrer lorsqu'il tentait d'apprendre et sur les défauts qu'elle a pu relever. Nous avons donc appliqué ses modifications au premier design pattern.

Nous avons ensuite conçu une deuxième version du prototype basé sur les premiers résultats. Ce deuxième prototype présentait bien sûr un design pattern différent. Nous souhaitions renouveler l'expérience à zéro avec notre cobaye. Cette personne a donc répété le même exercice que précédemment et nous a fait un second retour.

Nous avons répéter une troisième fois cette manipulation en appliquant les modifications au premier et au second design pattern. Le troisième prototype présentait alors un dernier design pattern. Nous avons donc soumis une troisième fois notre prototype à notre cobaye.

La manipulation aurait pu être répété encore de nombreuse fois, néanmoins elle aurait nécessité plus de temps. Dans l'idéal nous aurions pu effectuer 23 tests différents, pour les 23 design patterns. Grâce à cette collaboration nous avons réussi à atteindre des résultats concluants. La connaissance a pu être correctement transmise.

3.4 Recueil

Depuis le début de ce projet de recherche et d'innovation nous parlons d'une solution d'apprentissage du nom de "Recueil". Cette solution devait se construire en deux parties, le fond que nous avons vu précédemment (notre prototype) et une forme. Cette forme nous allons la découvrir dans ce chapitre.

Comme expliqué auparavant nous avons mené une enquête. Cette enquête incluait d'autres questions et l'une d'entre elles va nous permettre de déterminer quel support favoriser pour la forme de ce recueil.

Durant l'enquête nous avons demandé aux développeurs quel support d'apprentissage favoriser lors d'une formation autodidacte. La réponse qui ressort de ce sondage est largement devant les autres (cf annexe A.5) puisque les personnes interrogées ont répondu à plus de **63%** que le site internet leur semble être le support le plus adapté pour l'apprentissage.

Le site internet est donc le choix à favoriser. Or nous avons besoin d'apporter une touche d'innovation dans ce projet. Nous avons donc cherché une meilleure alternative aux sites internet. Nous avons trouvé une enquête qui révèle que l'utilisation des technologies mobiles dépasse depuis quelques temps l'utilisation des ordinateurs.

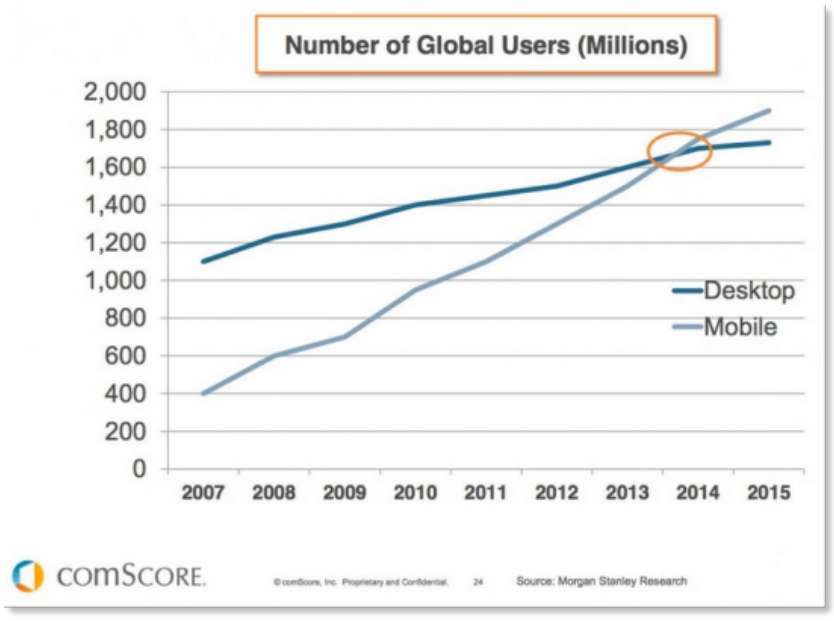

FIGURE 20 – Nombres d'utilisateurs global

En effet comme le montre cette étude[14], le nombre d'utilisateurs est en forte progression sur mobile, bien plus que sur ordinateur. Nous avons donc pensé qu'il pourrait être intéressant de favoriser les technologies mobiles qui regroupent les smartphones et les tablettes pour notre recueil. En plus de l'intérêt grandissant pour les technologies mobiles, nous visons ces technologies pour de multiples raisons.

Tout d'abord il est important de savoir que de plus en plus de solutions d'apprentissage prennent jour sur les technologies mobiles. Évidemment les utilisateurs sont plus intéressés d'apprendre confortablement installés chez eux qu'assis devant un ordinateur. De plus les technologies mobiles offrent une meilleure interactivité et ergonomie que les sites internets. Un aspect des plus importants pour une solution qui se veut visuelle.

Imaginons une réunion avec l'équipe d'architectes. Ils viennent nous soumettre une architecture constituée d'un design pattern. Plutôt que de mettre un terme à la réunion en attendant que nous maîtrisions le design pattern dont ils parlent (parce que nous ne les avons simplement plus en tête), il nous suffit simplement de démarrer notre application afin de réviser rapidement et ainsi prendre part à la réunion.

Pour permettre ceci l'application devra se découper de la manière suivante :

- Première partie, la vulgarisation. Une approche fondamentale du design pattern afin de se remémorer son utilité ;
- Seconde partie, la schématisation. Une approche schématique du design pattern afin de visualiser son agencement ;
- Troisième partie, la codification. Une approche technique du design pattern afin de comprendre son implémentation.

Nous avons pensé permettre à l'utilisateur de sélectionner le langage informatique qu'il préférait voir apparaître dans la partie "Codification". Dans le cas de notre prototype le code était en Java, mais dans la solution finale nous souhaitions permettre de choisir. Nous avons aussi souhaité ajouter un test pour permettre à l'utilisateur d'éprouver ses connaissances sur les design patterns GOFs. Nous pensons qu'il est toujours intéressant de pouvoir évaluer sa maîtrise sur un sujet. Et pour finir nous voulions permettre à l'utilisateur la transmission du code du design pattern vers sa boite mail. Ainsi l'utilisateur peut le récupérer depuis son poste de travail pour finir par l'intégrer dans son projet.

4 Conclusion

Depuis plusieurs années, mon intérêt s'est porté sur la programmation orientée objet, ce qui a créé une motivation en moi pour devenir un jour 'un architecte logiciel'. De nombreuses fois cet intérêt s'est également porté sur la différence qui subsistait entre le métier de technicien et d'ingénieur. Rapidement la conclusion s'est tourné vers l'analyse, à travers des concepts comme l'UML et les design patterns.

Mon travail était principalement consacré à ces sujets, que ce soit à travers mon PRI ou les différents projets sur lesquels j'ai pu travailler. Ma motivation à comprendre a soulevée de nombreuses questions sur le sujet et a aussi permis de multiplie débats.

A cette époque peu de personnes autour de moi cherchaient à prendre ce recul sur le métier de développeur. Un certains nombre de questions me son apparus. Pourquoi connaissaient-elles peu de choses sur l'UML et les design patterns malgré notre formation en commun ? Étaient-elles incapables d'aller au bout de cet apprentissage ? Se donnaient-elles à la possibilité de prendre du recul sur leur propre travail ?

Dans l'état de l'art nous avons vu l'UML, son origine, ses nombreux diagrammes et les différents outils qui permettent sa construction. Ensuite nous avons mené une enquête qui permettrait de répondre à toutes les hypothèses soulevées. Nous avons également étudié différents design patterns dans le cadre de la conception d'un prototype. Et finalement nous avons pensé le recueil sur sa forme et sur son fond.

Ce projet était pour moi une belle opportunité de travailler sur un sujet qui méritait qu'on accorde un peu de temps. Mon intérêt s'est porté sur la difficulté de maîtriser l'exercice pour lequel nous avons eu un grand intérêt, mais également de savoir accessibiliser cette connaissance par la suite.

Le but que nous nous sommes donnés dans ce projet de recherche et d'innovation était de créer un pont. Un pont entre ceux qui aiment échanger et réfléchir sur les méthodes de développement et ceux qui souhaitent faire de même sans y parvenir.

Ce pont durant ce projet a pris le nom de solution pédagogique, ou de recueil, puis d'application, et aussi de prototype. A travers ce projet l'objectif était de per-

mettre à ces personnes de s'élever au même niveau de réflexion. De cette manière, si ces personnes souhaitent prendre le temps d'échanger et de réfléchir ils le pourront à travers des connaissances qui permettent ce genre d'échanges.

"Si vous ne pouvez expliquer un concept à un enfant de six ans, c'est que vous ne le comprenez pas complètement." – Mark Twain

5 Glossaire

- **AGL** : (Atelier de génie logiciel) ensemble de programmes informatiques permettant eux-mêmes de produire des programmes de manière industrielle.
- **Classe abstraite** : Une classe abstraite est une classe possédant des méthodes abstraites.

  Une méthode abstraite ne possède pas d'implémentation, mais est utilisée par la classe elle-même ou une autre classe. Il n'est pas possible de créer et d'utiliser une instance directe d'une classe abstraite. Il faut utiliser une sous-classe qui implémente toutes les méthodes abstraites.
- **Diagramme** : Schéma graphique offrant un support de réflexion pour les différentes phases d'analyse lors d'un projet de réflexion. Il synthétise donc le fonctionnement de l'un des aspect du projet.
- **Design Pattern** : Patron de conception ou schéma, retrouver principalement dans les diagrammes de classes. Constitué d'un ensemble d'entités il à pour but de répondre à un besoin bien spécifique. Un Design pattern va donc souvent répondre à un problème donnée par sa solution prédéfinie.
- **Encapsulation** : Le principe d'encapsulation est un concept qui permet de cacher un ensemble d'informations derrière un principe de privatisation (public/privé/protégé). Ces données se veulent accessible à l'aide d'un accesseur ou d'un mutateur.
- **État de l'art** : État d'un ensemble de connaissances à un instant donné.
- **Gang of Four** : (*Gang des quatre* en français) est un pseudonyme accordé aux auteurs du livre "Design Patterns : Elements of Reusable Objet-Oriented Software". Le GOF est constitué d'Erich Gamma, de Richard Helm, de Ralph Johnson et de John Vlissides. On assimile parfois le terme GOF au 23 Design Patterns issue de leur livre.
- **GNU GPL** : (General Public license) Licence publique général GNU est une licence imposant un ensemble de conditions légales de distribution des logiciels libres du projet GNU.
- **Interface** : Une interface ne contient que des méthodes abstraites. Comme une classe abstraite, il n'est pas possible de créer et d'utiliser une instance directe d'une interface. Il faut utiliser une classe qui implémente cette interface, c'est à dire toutes les méthodes abstraites de celle-ci.

- **JPEG** : Joint Photographic Experts Group, extension de fichier d'image populaire sous Windows. Connu aussi sous l'extension de .jpg, .jpeg, .JPG, .JPEG.
- **Licence EPL** : Eclipse Public Licence
- **OMG** : Object Management Group est une organisation international issue d'un consortium à but non lucratif. Groupe en charge du suivi et de la standardisation d'un ensemble de projet du même ressort que l'UML.
- **Outils de modélisation UML** : Logiciel couramment utilisé durant les phases d'analyse et de développement dans les projets d'informatiques. Il permet la modélisation des différents diagrammes UML.
- **POO** : (Programmation Orientée Objet) est un paradigme de programmation qui se base sur la présence d'objets. Cet objet représente une entité constitué d'un corps chargé d'instructions.
- **PRI** : Projet de recherche et d'innovation. Projet ayant lieu en 4e et 5e année durant la formation de Manager en Système d'informations à l'Exia.CESI.
- **RTF** : Revision Task Force, chez l'OMG ils sont l'équipe en charge de l'évolution du produit UML.
- **Thread** : Un thread (processus) est une entité indépendante chargé d'effectuer un ensemble d'instructions. Généralement les threads fonctionnent en parallèle pour éviter de bloquer le déroulement d'un programme.
- **UML** : Langage de modélisation graphique constitué de pictogrammes. Utilisé principalement dans le projet d'informatique.

Références

[1] Wikipédia. Uml (informatique) — wikipédia, l'encyclopédie libre, 2014. [En ligne ; Page disponible le 28-janvier-2014].

[2] Philippe Kruchten. *Architectural Blueprints - The "4+1" View Model of Software Architecture*. IEEE Software 12 (6), 1995.

[3] Tim Weilkiens and Bernd Oestereich. *UML 2 Certification Guide*. Denise E. M. Penrose, San Francisco, 2007.

[4] OMG.org. Unified modeling language™ (uml®), 1997-2014.

[5] Uml-diagrams. http ://www.uml-diagrams.org/, 2007-2014.

[6] Center for Environmental Structure. http ://www.patternlanguage.com/, 2013.

[7] Richard Helm John Vlissides Erich Gamma, Ralph Johnson. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[8] Uml en français. http ://uml.free.fr/index.html, 2015.

[9] Developpez.com. Sondage : Quel outil de modélisation uml utilisez-vous ?, 2007-2014.

[10] Pour les nuls. http ://www.pourlesnuls.fr/, 2015.

[11] Object Oriented Design Design Patterns. http ://www.oodesign.com/.

[12] Les motifs de conception (design Patterns). http ://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm, 1997-2014.

[13] Design Pattern & Refactoring. https ://sourcemaking.com/, 2015.

[14] Danyl Bosomworth. Mobile marketing statistics 2015.

A Annexe

A.1 Sondage internet

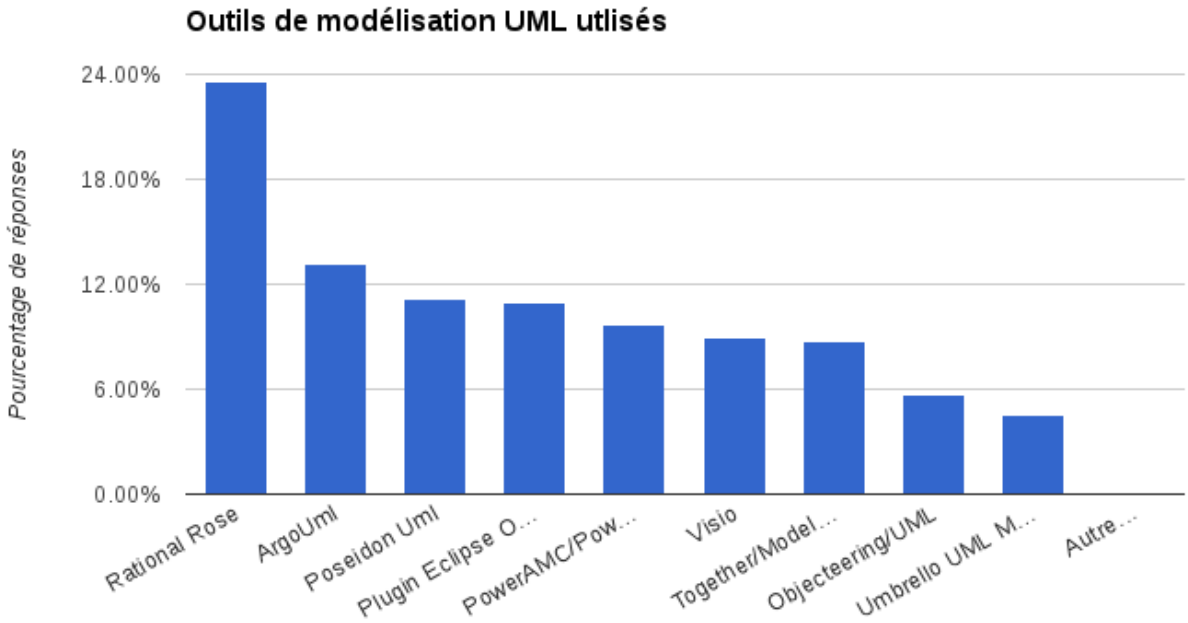

FIGURE 21 – Sondage UML

A.2 Enquete 1

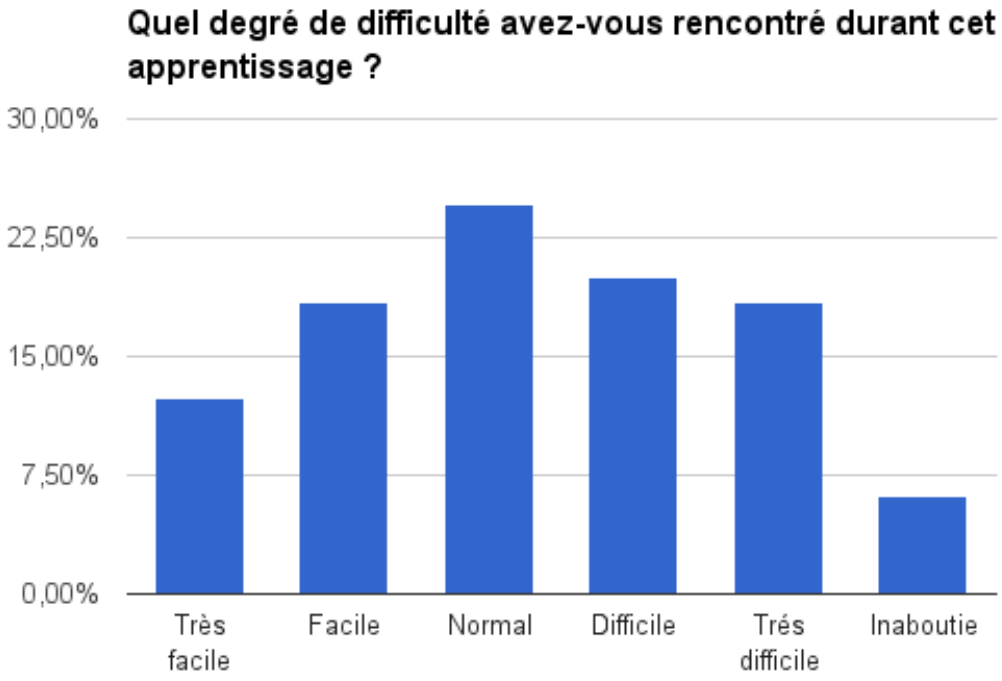

FIGURE 22 – Difficulté d'apprentissage

A.3 Enquete 2

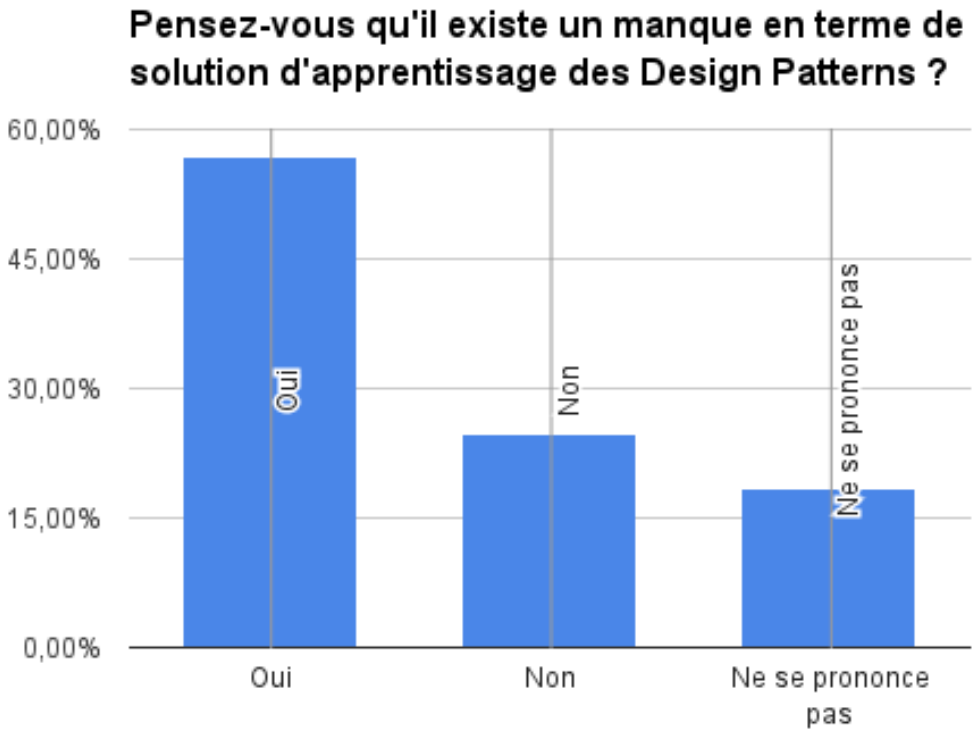

FIGURE 23 – Manque de solution d'apprentissage

A.4 Enquete 3

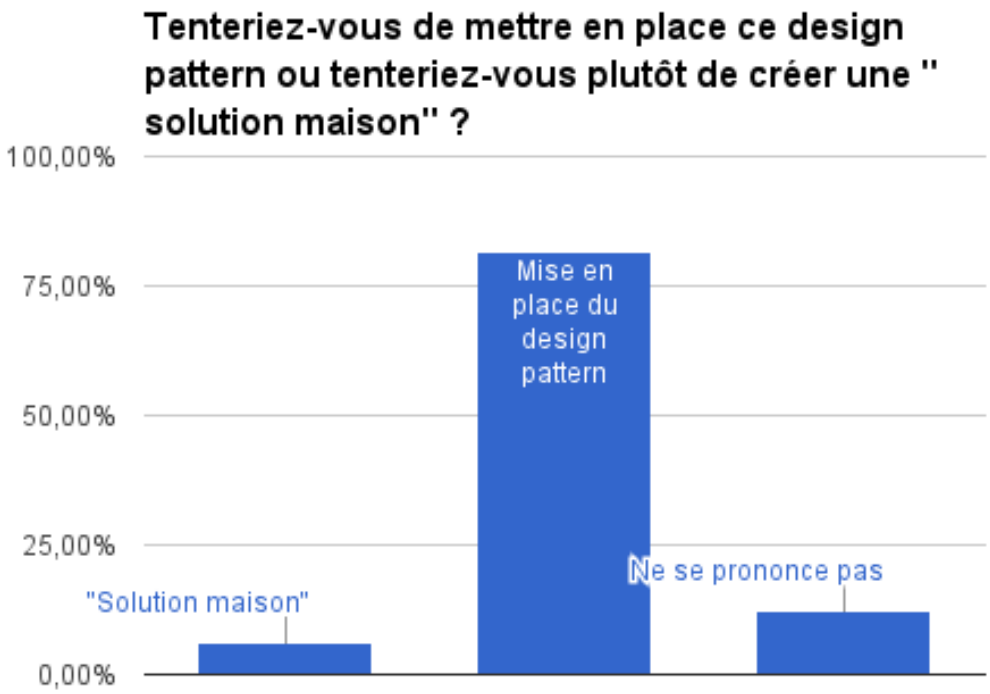

FIGURE 24 – Mise en place d'un design pattern

A.5 Enquete 4

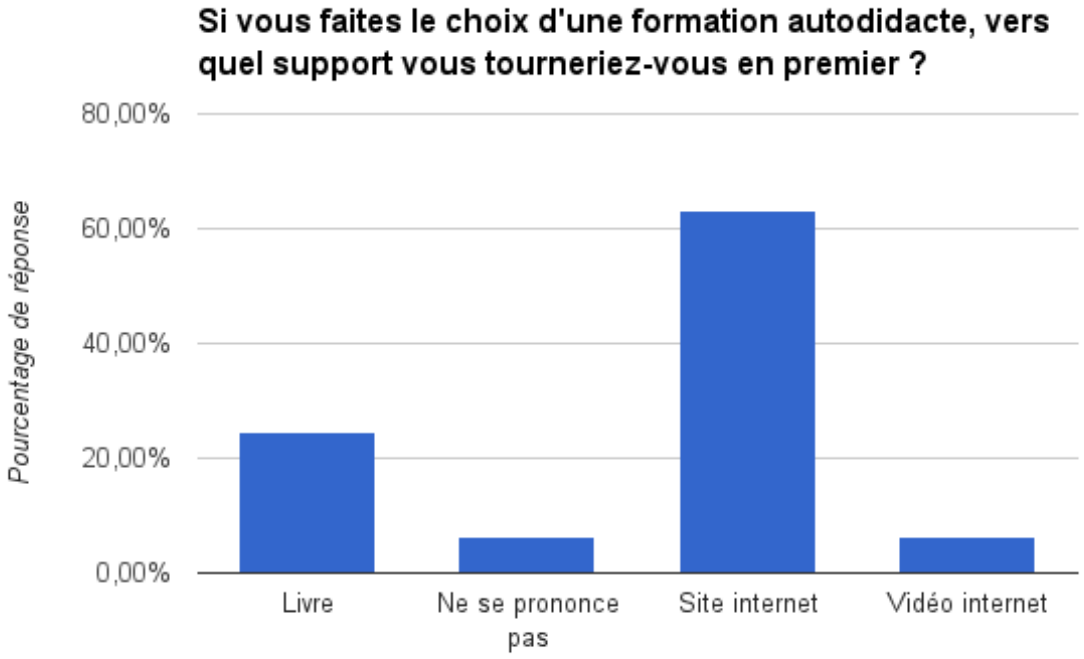

FIGURE 25 – Support favori d'apprentissage