Le prototypage basé sur des méta données

phase 1 du cycle de développement

Mémoire

Dario Gomez

Maîtrise en sciences de l'administration
Maître ès sciences (M.Sc.)

Québec, Canada

Résumé

Le processus de conception des systèmes d'information (SI) est un processus long et complexe qui résulte en de nombreux échecs. Le prototypage informatique et la conception guidée par modèles ont été proposés comme une solution en améliorant la qualité des spécifications au début du cycle de vie d'un SI.

L'objectif de notre recherche est de mieux orienter l'action de spécification des exigences dans la phase initiale de conception « Communication Client - Concepteur » et dans le début de la phase de développement « Communication Concepteur - Développeur » en utilisant des artefacts de prototypage.

Notre travail ouvre concrètement une voie dans laquelle il devient possible d'envisager que toute modification au cours de la vie d'un SI puisse être effectuée à partir du modèle du domaine qui est l'intrant du « prototypeur », qui devient alors le SI lui-même.

Mots clés: système d'information; méthodologie de conception; modèle conceptuel de données; spécification déclarative; spécification exécutable; prototype; méta-donnée; architecture applicative

Abstract

Designing information systems is a lengthy and complex process that leads to numerous failures. Prototyping has been proposed as a solution to improve the specifications' quality in the beginning of an information system's life cycle. Every information system (IS) is based upon the information architecture ; it is, before all, a content about the perceived reality. A "domain" is a formalization of the perceived reality in which the IS users identify the representations of facts (the data) by means of semantic keys. IS designers have to transform this model using their knowledge about the abstract functioning of computers.

The objective of our research is to guide the action of requirements specification in the initial design phase of " Communication Customer - Designer" and in the beginning of the development phase "Communication Designer - Developer" using prototyping artifacts.

Our work actually opens the way where it becomes possible to envisage that every modification during the information system's life cycle could be done from within the domain model, which is an input for the "prototyper" and becomes then itself an information system.

Keywords : information system ; design method ; conceptual data model ; déclarative specification; executables pecification; prototype ; méta-data ; application architecture

Table des matières

Liste des tableaux

Liste des figures

A mi madre, que ya no esta para
acompañarme.
A mis hijas y mi padre que
siempre me han acompañado.
$\therefore$

Ainsi, le peintre qui désire qu'un certain lieu de son tableau soit de couleur verte, y place un arbre ; et il dit par là quelque chose de plus que ce qu'il voulait dire dans le principe.

(Valéry 1921)

Remerciements

"Écoute, Phèdre (me disait-il encore), ce petit temple que j'ai bâti pour Hermès,
à quelques pas d'ici, si tu savais ce qu'il est pour moi !
— Où le passant ne voit qu'une élégante chapelle,
— c'est peu de chose : quatre colonnes, un style très simple,
— j'ai mis le souvenir d'un clair jour de ma vie. O douce métamorphose !"
(Valéry - 1921)

Beaucoup de gens ont contribué à ce projet de recherche et à la réalisation de ce mémoire. Je suis très reconnaissant à tous ceux qui l'ont rendu possible. Tout d'abord, je tiens à remercier mon conseiller et mon point de repère tout au long de ce projet, le professeur Daniel Pascot, qui m'a permis de faire partie de l'équipe de recherche, et qui a rendu ce projet viable. Ses conseils et son point de vue perspicace et critique sur le développement de produits logiciels et l'analyse de données, ont été d'une immense aide et très instructifs. M. Pascot a su prendre le temps pour discuter du projet et me guider pour me permettre de mener à bien la recherche. D'autre part, j'ai eu la chance de bénéficier des apports des professeurs Dzenan Ridjanovic et Sehl Mellouli, leur participation active et leurs commentaires ont contribué significativement à la réalisation de ce projet.

Je voudrais aussi remercier le Centre de recherche et transfert d'architecture d'entreprise (CeRTAE), qui m'a été d'un support inconditionnel. Particulièrement à madame Gabriela Nino, pour sa grande patience et ses commentaires très opportuns au moment de tester les différentes versions du prototypeur. Également à monsieur Balla Diop, monsieur Sidi Mahmoud et à monsieur Aleksey Zolotaryov pour leur constant soutien et recommandations.

Enfin, je tiens à remercier également mes filles, Andréa et Nicole pour leur soutien et l'encouragement tout au long du cursus scolaire et du travail de mémoire. Et merci à tous les membres de l'équipe qui directement ou indirectement ont pu m'aider à matérialiser ce projet.

Introduction

« Simplicity is prerequisite for reliability. »
(Dijkstra 1975) [59]

0.1 Problématique

Le s systèmes d'information (SI) sont devenus une composante essentielle de nos organisations, ils sont incontournables pour soutenir et contrôler les processus d'affaires, la production et presque tous les aspects des organisations. Ainsi, la qualité de ces systèmes devient d'une importance capitale, car même leurs plus petits défauts peuvent avoir des impacts énormes sur la réussite des organisations.

Depuis 1968 le terme « Crise du logiciel » a été utilisé dans la littérature à plusieurs reprises[13, 58, 43, 67, 48, 160, 63, 153] pour exprimer les aspects suivants :

- souvent les SI réalisés ne correspondent pas aux besoins des utilisateurs ;
- les SI contiennent trop d'erreurs (qualité du logiciel insuffisante) ;
- les coûts du développement sont rarement prévisibles et sont généralement prohibitifs ;
- les délais de réalisation des SI sont généralement dépassés.
- la maintenance des SI est une tâche complexe et coûteuse ;

Par exemple, Le et al (2006) situent le taux d'échec des systèmes ERP dans l'ordre de 40 à 60 % [109] alors que Themistocleus et Irani (2001) reportent que 70% des implémentations ERP n'atteignent pas les bénéfices escomptés. Plus récemment, Mamoh et al. (2010) constatent que 51% des implémentations ERP sont perçues comme des échecs [118]. Ce qui nous amène au constat qu'il s'agit d'un problème encore non totalement résolu.

0.2 Les causes

Selon de nombreux auteurs[140, 60, 88, 41, 44] depuis les débuts de l'informatisation des organisations, il y a deux grandes étapes essentielles communes à tous les développements des

systèmes informatiques, indépendamment de leur taille ou de leur complexité. Il s'agit d'une première étape d'analyse [1], suivie d'une seconde étape de codage. Toutefois, la mise en ouvre des SI sur la base de ces deux étapes est souvent sanctionnée par un échec (Crise du logiciel)[140]. Brooks (1987) affirme que la principale difficulté, qu'il a appelée « essentielle », est la conception et non la construction, et que les erreurs de construction ne sont pas comparables à celles de conception[44]. Il soutient que la plupart des erreurs proviennent de la phase initiale de l'analyse, notamment à cause d'une mauvaise perception des besoins ou leur évolution due aux changements dans les environnements internes ou externes. Il soutient que dans la plupart des cas, les clients ne savent pas ce qu'ils veulent de la part du système d'information (bien qu'ils sachent ce qu'ils font en pratique), qu'ils n'ont presque jamais pensé à tous les détails nécessaires pour spécifier ce dont ils ont besoin de la part de leur système d'information informatisé. Il est, en effet, quasiment impossible d'indiquer de façon complète, exacte et correcte les exigences d'un système informatique moderne[44]. Bien entendu, lors de la spécification des exigences, on manque souvent de l'information nécessaire pour concevoir certains aspects du système. Dans de tels cas, il est laissé aux programmeurs l'interprétation des exigences et la découverte des « solutions », ce qui peut souvent conduire à des solutions qui ne répondent pas aux besoins[25] et les conduit à bien résoudre un mauvais problème. Frost et Campo (2007) ont confirmé l'impact négatif que pourrait avoir des changements dans les différentes phases d'un projet [76]. Tel qu'indiqué au Tableau 0.1, il apparaît clairement que les coûts des changements augmentent de manière importante au fur et à mesure que le projet avance.

Requirement	Design	Coding	Development Test	Acceptance Test	During Operation
1	3-6	10	15-40	30-70	40-1,000

TABLE 0.1: Temps nécessaire pour effectuer des changements dans les différentes phases d'un projet d'informatisation [76]

0.3 Pistes de solution

On a identifié le prototypage[88, 136, 35] et la conception guide par modèles (CGM)[152, 143, 39, 65, 27, 71] comme les pistes de solution les plus récurrentes dans la littérature. Pourtant, jusqu'ici aucune de ces solutions a réussi à apporter une solution convenable à la crise du logiciel sur laquelle il y ait unanimité.

1. LeMoigne fait la différence entre l'analyse (disjoindre pour réduire sans ambigüité à des éléments simples) et la conception ("construire dans sa tête avant de construire dans la ruche").[101].

0.3.1 Prototypage

Pascot (2001) affirme qu'une des principales raisons d'échec dans l'implantation des systèmes d'information est liée au fait que la plupart des méthodes actuelles de conception repoussent la prise en compte des importants détails concernant la définition des informations aux dernières étapes de la conception et même dans la réalisation. Il ajoute qu'une solution classique à cette problématique est proposée avec le recours à des « prototypes » car ils offrent la possibilité de valider les spécifications fonctionnelles détaillées concernant les informations au tout début du projet [130]. Lim et al. (2008) ont donné un excellent aperçu des raisons pour le prototypage. Ils déclarent que le prototypage est une activité dans le but de créer une manifestation qui, dans sa forme la plus simple, filtre les qualités auxquelles les concepteurs sont intéressés, sans fausser la compréhension de l'ensemble[110].

L'idée du prototype n'est pas nouvelle, en 1970, Royce écrivait : «It is simply the entire process done in miniature, to a time scale that is relatively small with respect to the overall effort (do it twice) »[140, Page 334]. La figure 0.1. représente la méthode connue sous le nom « cascade » (waterfall) développée par Royce en 1970 et utilisée encore dans nos jours dès que les projets de toute nature sont de grande envergure dans laquelle s'insère le recours aux prototypes.

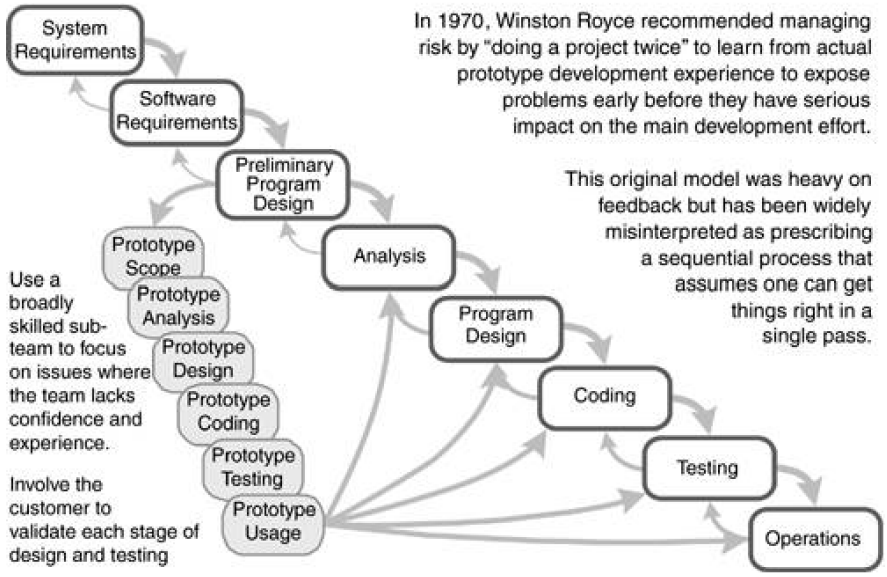


FIGURE 0.1: Méthode de développement en cascade original [136]

Boehm fait la promotion de l'utilisation de prototypes pour diminuer le risque. Le prototypage est le fondement de sa méthode de développement en spirale (figure 0.2). Il stipule que le « Prototypage » est utilisé avec succès pour acquérir une compréhension rapide des exigences du système et pour guider ou encadrer la conception de logiciels. Il précise que cela sert aussi à évaluer les interfaces utilisateur et à tester les algorithmes complexes. Quand les utilisateurs voient un résultat tangible sur l'écran, ils peuvent comprendre ce qui manque ou ce qui pourrait

être la prochaine question. Les expériences de Horowitz et Boehm ont montré qu'avec le prototypage il est possible dans certains contextes de réduire l'effort de développement de 40%[88, 136, 35].

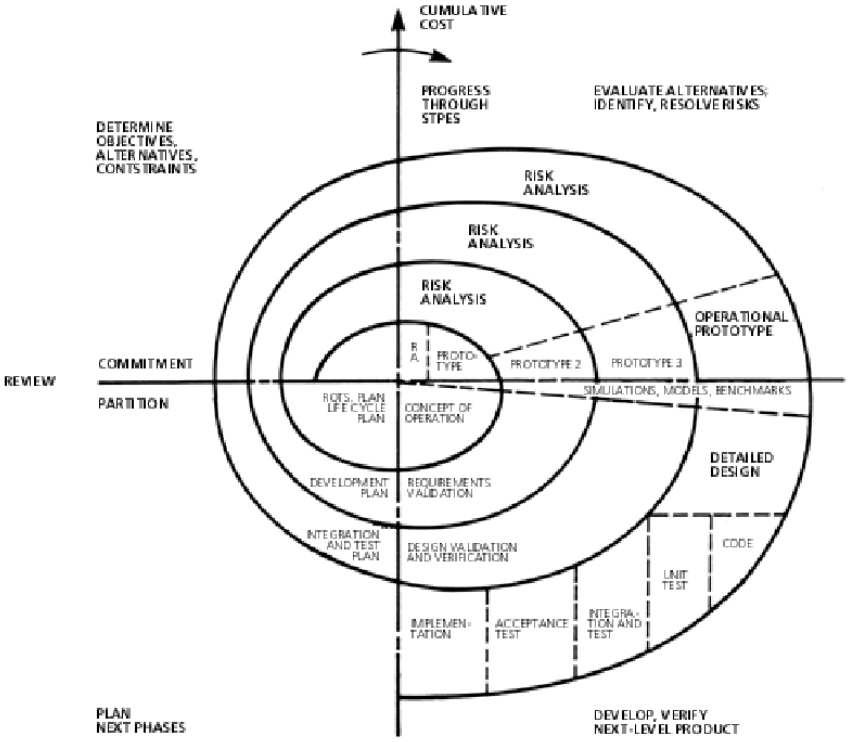


FIGURE 0.2: The « Spiral Model »[41, P64]

Bernstain (1996) affirme que "For every dollar invested in prototyping one can expect a $1.40 return within the life cycle of the system development." [35]

Ainsi, la valeur du prototypage dans la mise en œuvre des systèmes d'information est clairement reconnue par de nombreux auteurs[130, 36, 35, 25, 158, 20]. Les avantages reconnus aux prototypes sont les suivants :

- moyen très efficace pour :
  - mieux comprendre et valider les besoins,
  - réduire la complexité du problème (en rejetant à plus tard ce qui concerne la sécurité, la performance, les contraintes héritées du passé, ...)
  - fournir une validation précoce des fonctionnalités du système.
- facilitation de la communication dans les étapes successives entre le client et le concepteur

- conduit à des spécifications plus créatives et un système plus tourné vers l'avenir.
- démonstration de ce qui est fonctionnellement et économiquement réalisable.
- permet une approche évolutive : « échouer petit » afin qu'il puisse «réussir grand".

Un important principe du prototypage indique que le client est le coauteur, le prototypage implique la collaboration du client, tel qu'est défini par le manifeste AGILE « Customer collaboration over contract negotiation. » [31]. Schrage (2004) résume bien la question quand il affirme «Clients are prototyping partners, not customers. We model, prototype, and simulate software with clients, not for them » [144].

Carr et Verner (1977) affirment que les modèles et les spécifications écrites peuvent cacher toutes sortes de défauts, mais quand les clients sont réellement assis en face d'un prototype et l'utilisent, les défauts deviennent évidents à la fois en terme d'erreur et d'exigence mal comprise[45]. Il est cependant évidant à travers la littérature que le prototypage n'est pas un « balle d'argent, alias silver bullet » [2]. Mais, il est aussi clair que c'est un outil potentiellement efficace dans la réussite de mise en place des systèmes d'information.

La construction des prototypes est en soi un projet de construction d'un système d'information et la production des systèmes d'information est une série d'activités intellectuelles et techniques réalisées par des ingénieurs hautement scolarisés, il reste que malgré les promesses de réussite, la première impression est souvent que le prototype augmente le temps et les coût nécessaire pour le projet final[110][24]. En conséquence, le prototypage, bien que considéré par certains comme une solution théorique idéale, a toujours été sous-utilisé en pratique à cause des efforts et des coûts de construction supplémentaires [136][25].

0.3.2 Conception guidée par modèles

> « Smart data structures and dumb code works
> a lot better than the other way around »
> (Raymond 2005) [137]

La conception guidée par le modèle (CGM) est une approche de construction des SI qui vise un passage plus direct à partir de la modélisation vers la production des SI[152]. Il y a plusieurs variantes à cette approche, notamment :

- Model-driven engineering **MDE**[143] ;
- Model-Driven Architecture **MDA**[39] ;
- Domain Driven Design **DDD**[65]

2. Les balles d'argent sont des objets magiques capables de tuer les monstres comme le loup-garou entre autres. Le terme est devenu familier dans le contexte des SI suitese à l'article de Frederick P. Brooks, Jr. qui fait l'analogie des SI qui peuvent devenir des monstres terrifiants d'un moment à l'autre[44].

– Model-Driven Development **MDD**[27].

En général l'approche CGM est basée sur l'idée que les modèles sont les artefacts primaires (fondateurs) à partir desquels d'autres artefacts sont générés[71]. Les caractéristiques communes sont :

– Le modèle doit être lisible et interprétable par l'ordinateur. Le modèle n'est pas un simple graphique même s'il peut être affiché comme tel.

– La lisibilité et l'interprétation des modèles sont des conditions préalables pour être en mesure de générer des artefacts.

– Pour interpréter les modèles, il est nécessaire définir un langage, ce langage est aussi un modèle qui est appelé métamodèle.

«Meta» signifie littéralement «après» en grec. En informatique, le terme est largement utilisé avec plusieurs significations différentes[119], entre autres :

– pour les données, "méta-donnée" signifie donnée sur les données et réfère aux référentiels des bases de données, d'une manière générale "méta-chose" en informatique est ce qui contient la chose de façon à ce qu'elle puisse être l'objet de manipulation par un programme dans un ordinateur ;

– dans la modélisation conceptuelle, "méta-modèle" est un modèle d'un modèle, le méta-modèle lui-même est à la fois un modèle et un langage qui sert à définir les composantes, les structures et les règles du modèle. Les méta-modèle sont à la base de la CGM.

Ainsi, le terme méta-donnée souvent est utilisé de façon générique pour indiquer une structure qui définit le gabarit des autres structures.

Dans CGM, l'idée centrale de construction des SI comme l'assemblage d'objets est progressivement remplacée par la notion de transformation de modèles[65] jusqu'à ce que le modèle devienne la structure d'un système en production. Par exemple le modèle physique de base de données, un écran de saisie ou un rapport imprimé. Cette approche, bien que pas encore universellement acceptée comme une pratique courante [69], est perçue de plus en plus [27] comme une possible solution à la crise du logiciel.

La CGM fait partie des méthodes de spécification déclarative[151]. Ce sont des méthodes de conception des SI qui mettent l'accent sur le «quoi» (la règle) et « que » faire (le résultat), elle s'opposent à la déclaration procédurale ou impérative qui porte sur le «comment» faire (la pocédure), c'est-à-dire la structure de contrôle des étapes nécessaires pour atteindre la solution. Par exemple, le langage HTML est « déclaratif » car il décrit ce que contient une page (texte, titres, paragraphes, etc.) et non comment les afficher (positionnement, couleurs, polices de caractères, etc.).

La philosophie de CGM est basée sur l'idée que les modèles doivent être utilisés pour générer directement des systèmes exécutables[151]. Forward et Lethbridge (2008) on observé que 68% des participants considèrent que le plus gros problème des approches CGM est l'impossibilité

de garder le modèle à jour avec le code généré et que seulement 14% des répondants utilisaient des outils pour générer le code à partir de modèles[71].

L'approche CGM dans ses différentes variantes est un option intéressante pour solutionner la crise de logiciel. Pourtant, il y a des écueils dans la pratique [37], par exemple :

- Les langages de déclaration ne sont pas suffisamment complets pour spécifier tout ce qu'il faut dans un système informatique.
- Les frameworks rendent normalement très difficile d'étendre le logiciel au-delà de la partie automatisée.
- Les détails d'implantation qui sont ajoutés après la génération du code sont perdus s'ils ne sont pas remontés au modèle.

Par exemple, MDA[123] [3], une des principales propôsitions dans le monde CGM a été introduit en 2001, et aujourd'hui (2013) l'industrie de logiciels ne l'a pas adoptée massivement [148]. À propos de cela, Evans (2004) le créateur de la méthode DDD (Domain Driven Developpement)[65], soutient que si UML (Unified Modeling Language) est un excellent outil, il n'est pas suffisant pour tous les cas, et il faut retourner à des schémas et des textes pour assurer une bonne communication. Donc, la génération automatique à partir de UML n'est pas une option totalement satisfaisante. En résumé, bien que la CGM n'est pas encore prête pour la production automatique des SI, il est aussi vrai que les principes d'automatisation des processus de basés sur de méta-modèles sont reconnues[37, 54, 148, 28, 143, 20, 22, 25, 70, 125] comme un véritable pas en avant pour avancer dans la solution de la crise de logiciel.

0.4 Question de recherche

Le processus de conception des systèmes d'information (SI) est un processus long et complexe qui résulte en de nombreux échecs (crise du logiciel). Le prototypage informatique a été proposé comme une solution en améliorant la qualité des spécifications au début du cycle de vie d'un SI.

Le prototypage, bien que considéré par certains comme une solution théorique idéale, a toujours été peu utilisé en pratique à cause des efforts (coûts et temps) de construction. La conception guidée par le modèle (CGM) est une approche de construction des SI qui vise un passage plus direct à partir de la modélisation vers la production des SI. En théorie, cela serait une solution à la crise du logiciel. Pourtant, les technologies et méthodes CGM ne sont pas encore assez matures pour une adoption massive.

Avec les idées ci-dessus, nous soutenons que sur le plan conceptuel, que bien la CGM n'e soit pas encore mature, il est possible d'utiliser cette approche pour mieux orienter la création

3. MDA est chapeauté et basé sur des standards l'OMG "Object Management Group" comme UML, MOF, XMI, entre autres.

des prototypes à partir des modèles, et ainsi intégrer le prototypage basé sur CGM comme un élément central des approches traditionnelles de conception des SIO. Ainsi, l'objective de notre recherche est spécifiquement de mieux orienter l'action de spécification des exigences dans la phase initiale de conception « Communication Client - Concepteur » et dans le début de la phase de développement « Communication Concepteur - Développeur » en utilisant des artefacts de prototypage.

Cela nous amène à notre question de recherche :

- Comment la CGM peut être utilisée pour construire des prototypes de SI transactionnels (SIT) ?

L'objectif de cette approche étant de maximiser la rapidité et l'efficacité de la validation des exigences, va conduire à la proposition d'un méta-modèle qui intègre les caractéristiques plus importantes destinées à faciliter la communication « Client - Concepteur » et « Concepteur - Développeur ». La contribution pratique de notre recherche consiste à développer un outil qui soit capable d'interpréter la méta structure et produire un prototype fonctionnel des SIT en suivants les directives de la CGM.

Présentation des chapitres

- Chapitre 1 : Revue de la littérature sur le sujet de génération de prototypes et l'approche CGM. Dans notre recherche on fait la distinction entre les travaux qui utilisent un prototype pour démontrer un concept et ceux qui ont pour objectif la génération d'un prototype fonctionnel d'un système d'information transactionnel informatisé (SIT).
- Chapitre 2 : Fondements relatifs à la conduite d'un projet d'informatisation, présente un aperçu des concepts et outils pertinents qui permettent d'atteindre notre objectif. Dont, la philosophie de la modélisation et l'importance du prototypage dans la conception des systèmes d'information.
- Chapitre 3 : Spécification déclarative, analyse l'état de l'art et des tendances de la technologie informatique en construction de systèmes d'information ainsi que les zones où on peut appliquer notre recherche.
- Chapitre 4 : Évolution du projet, présente un aperçu des expériences précédentes et des étapes suivies pour la réalisation de ce travail.
- Chapitre 5 : Évolution du projet, définit les objectifs de développement et fournit les bases méthodologique et d'architecture utilisées dans ce travail.
- Chapitre 6 : L'interface de l'application Prototypeur, présente les éléments généraux de l'interface et introduit les concepts d'utilisation de l'application développée.
- Chapitre 7 : Démarche de Prototypage, propose une méthodologie de conception basée sur le prototypage et présente les étapes requises pour la construction des prototypes.

- Conclusion, présente un aperçu des contributions de ce travail et suggère des pistes de recherche futures.
- Appendice A :Description du code utilisé pour la génération de prototypes.

Les chapitres 2 et 3 ont été compilés à partir de sources citées en ajoutant des remarques de l'auteur, tandis que les chapitres successifs représentent notre expérience et contribution originale.

Chapitre 1

Revue de la littérature

Ce chapitre est consacré à la revue de littérature sur les principales dimensions de notre recherche. Soit l'approche CGM ou la construction des prototypes SIT. La première partie de cette section présente un résumé des différents articles qui se sont attaqués à la même problématique avec les solutions proposées. Dans la deuxième partie, on fait appel aux méthodes connues pour faire un révision des points forts présentés par chacun des auteurs et le positionnement par rapport à notre approche. La dernière partie est une discussion sur l'originalité de notre approche.

1.1 Travaux Connexes

Weiner et al. [158] décrivent une approche pour générer dynamiquement des interfaces de base de basé des données sur le Web. Ils utilisent un méta-modèle qui contient des informations sur les éléments nécessaires pour représenter un schéma de base d'un modèle de données (noms de table, noms des champs, type de données, et des liens entre les tables). Le méta-modèle est très simple et il n'y a pas des possibilités de configuration autres qu'un liste avec tous les champs.

Elsheh et Ridley[64] ont proposé un modèle qui vise à générer des formulaires Web dynamiques basés directement sur les métadonnées extraites des tables système. Ils ont utilisé un outil java pour convertir les métadonnées extraites dans un document XML. Un deuxième pas fait la conversion vers un document XHTML qui est présenté sur le navigateur. C'est l'option la plus simple et la plus directe. Pourtant il n'y a aucun de possibilité de configuration. Les règles de définition de l'interface utilisateur sont construites dans le code de l'application.

Mgheder et Ridley**[114]** ont proposé une approche similaire à Elsheh et Ridley[64] avec des technologies différentes.

Albhbah et Ridley[18, 19, 20] proposent l'utilisation des règles en conjonction aux métadonnées pour la génération des formulaires Web automatiques. La nouveauté de son approche réside

en la séparation de deux couches : Les modèles des données et les modèles d'interface. Cette approche améliore l'accessibilité et permet plus de flexibilité et de contrôle.

Antovic et al. [25] proposent comme objectif l'identification des corrélations entre l'utilisation des cas d'utilisation, des modèles de données et des interfaces utilisateur. Dans cette étude, les auteurs ont introduit un méta-modèle pour exprimer les exigences logicielles. Sur la base de ce méta-modèle, il est possible de concevoir et de mettre en œuvre l'interface utilisateur et l'automatisation de certains processus. L'article est base sur l'hypothèse d'un modèle de données déjà prêt, ce qui laisse de côte toute la problématique de la construction des modèles de données. Par contre il est très pertinent dans la définition des cas d'utilisation et dans la variété des objets d'interface propose à l'utilisateur. L'approche de prototypage n'est pas bien décrite, car leur objectif est l'identification des corrélations et non la proposition d'un modèle spécifique pour générer des prototypes.

Forward et al. [70] proposent une approche qui utilise un langage de modélisation sous le nom UMPLE pour générer des prototypes exécutables. UMPLE est un langage de modélisation qui peut être spécifié graphiquement et textuellement. Il permet de créer des diagrammes de classes et des diagrammes d'état. Les modèles de l'interface utilisateur sont générés sur la base des modèles des données. Le principal avantage de cet outil est la simplicité de fabrication des spécifications d'entrée, tandis que le principal inconvénient est l'impossibilité de définir de façon spécifique l'interface d'utilisateur.

Ridjanovic[139, 138] a proposé une famille de logiciels open source sous le nom « Modelibra » qui est utilisé pour développer des applications web dynamiques basées sur des modèles de domaine. Ce projet est constitué d'un outil de conception graphique, un modèle de domaine, un cadre de composants Web, une collection de déclarations CSS et XML, une base de données dynamique non persistante et un générateur de code Java. La démarche de génération de prototypes de Modelibra concerne la génération des spécifications du modèle dans la forme des documents XML et la génération code Java pour l'exécution en utilisant le framework Apache-Wicket et des fichiers HTML et CSS.

1.2 Positionnement

Sur la base de cette étude de la littérature on a identifié les points plus importants qui contribuent à la solution de notre objectif de recherche :

- méta-modélisation
- Approche de génération de prototype

1.2.1 Méta-modélisation

Pour évaluer les apports des projets étudiés, on a utilisé la méthode « DataRun » [129, 128, 130]comme guide. La méthode Datarun préconise qu'on doit réaliser le plus tôt possible le modèle conceptuel des données (MCD) car toutes les tâches du SIT sont organisées autour et à partir de ce modèle de données. Ceci veut dire que la réalisation du modèle des données précède tout diagramme de traitement de l'information et donc du fonctionnement du SI, mais bien sûr elle implique au préalable la connaissance, et si possible la représentation, du fonctionnement de l'organisation. Les activités qui précèdent la réalisation du MCD doivent être limitées à ce qu'il est nécessaire de découvrir et de documenter pour sa réalisation. Datarun propose qu'à partir d'un MCD, on peut construire un modèle de spécification d'interface (MSI) qui sert de base à la définition des composants de haut niveau et qui permet l'expression de l'interface d'utilisateur.

Pour la définition du méta-modèle, il faut tenir en compte que du point de vue de l'utilisateur, la différence entre la couche d'interface (définie par le MSI) et les autres couches du SIT ne sont pas clairement distinguables[25]. En fait, il sont complètement invisibles. Pour la plupart des utilisateurs, le SIT représente un moyen pour accomplir certaines tâches. Pour eux, l'interface utilisateur n'est pas simplement une partie du système de logiciel, mais le système lui-même[156]. D'autre part, le concepteur du système perçoit l'interface utilisateur comme la réalisation de l'entrée et/ou sortie du SIT. Pour lui, le SIT est déterminé pour les données et les règles de gestion[156]. Donc, les deux aspects son profondément liés et il n'est pas possible de privilégier une pour l'autre.

1.2.2 Approche de génération de prototype

Lors de notre discussion sur CGM, on a brièvement mentionné l'idée de la transformation de modèles. Selon la littérature[53][150][74] il y a deux mécanismes pour passer d'un niveau de modélisation vers un autre plus bas (par exemple passer des modèles des structures de données vers les modèles d'interface). Soit la génération de code ou l'interprétation de modèles sont utilisées. Voici un résumé des avantages de ces approches l'une par rapport à l'autre.

Avantages de la génération de code

La génération de code présente les avantages suivants par rapport à l'interprétation de modèles :

- La mise en œuvre de la solution est plus facile à comprendre : On peut regarder le code généré pour comprendre directement le comportement d'une application. Dans les cas de l'interprétation de modèle, on doit comprendre la mise en œuvre générique de l'interpréteur et la sémantique du modèle.
- Il fournit un contrôle supplémentaire : lorsque vous générez du code, ce code peut être modifié.

– La mise au point du générateur lui-même est plus facile que la mise au point d'un interprète : La création d'un interprète demande des travaux supplémentaires.

Avantages de l'interprétation de modèles

L'interprétation de modèles présente les avantages suivants par rapport à la génération de code :

– Il permet des changements plus rapides : Les changements dans le modèle ne nécessitent pas une régénération explicite, reconstruire, répéter l'essai, et redéployer étape. Cela conduira à un raccourcissement significatif du temps de traitement.

– Il permet des changements à l'exécution : parce que le modèle est disponible à l'exécution, il est même possible de changer le modèle sans arrêter l'application en cours d'exécution.

– Plus facile à modifier pour la portabilité : un interprète en principe crée une cible indépendante de la plate-forme pour exécuter le modèle. Il est facile de créer un interprète qui fonctionne sur plusieurs plates-formes (par exemple, plusieurs systèmes d'exploitation, plates-formes dans les nuages).

– Plus facile à déployer : Lorsque la génération de code est utilisée, on voit souvent qu'on a besoin d'ouvrir le code généré dans Eclipse ou VisualStudio et compiler pour créer l'application finale. Dans le cas de l'interprétation de modèles, on a juste à lancer l'interprète et mettre le modèle en elle. Le code n'est plus nécessaire. Par conséquent, il est beaucoup plus facile pour les experts du domaine pour déployer et exécuter une application au lieu de seulement le modéliser.

– Plus facile à mettre à jour : Il est plus facile de changer l'interprète. On n'a pas besoin de générer le code à nouveau en utilisant le dernier générateur.

– C'est plus sécuritaire : Par exemple, sur une plate-forme de nuage (Cloud) il est suffisant de télécharger le modèle, il n'est pas nécessaire d'accéder au système de fichiers ou d'autres ressources système. Seul le code dans l'interpréteur peut accéder aux bibliothèques système. L'interprète fournit une couche supplémentaire au-dessus de l'infrastructure, tout en dessous est extrait de suite. C'est essentiellement l'idée de Platform-as-a-Service (PaaS).

En conclusion, nous avons observé que les raisons de préférer l'interprétation sont nombreuses et que, bien que cela peut signifier plus de travail, le résultat final sera beaucoup plus souple et pratique pour notre propos.

1.2.3 Analyse des travaux connexes étudiés

Aux termes de ce qui précède, on analyse les différentes approches pour déterminer lesquelles sont pertinentes pour l'atteinte de notre objectif de recherche.

L'approche la plus simple est celle de Weiner et al. [158]. D'un point de vue structurel, la table de méta-données contient des informations sur les quatre éléments essentiels nécessaires pour

Table Name	Field Name	Field Data Type	Linked Table Name	Linked Table Field
Table A	PK_FieldA1	Number	---	---
Table A	FK_FieldA2	foreign	Table B	PK_FieldB1
Table A	FK_FieldA3	Foreign	Table C	PK_Field C1
Table A	FK_FieldA4	foreign	Table D	PK_Field D1
Table A	FieldA5	Number	---	---
Table D	PK_FieldD1	Number	---	---
Table D	FK_FieldD2	Foreign	Table E	PKField E1
Table B	PK_FieldB1	Number	---	---
Table B	Field B2	String	---	---
Table B	Field B3	Number	---	---
Table C	PK_FieldC1	Number	---	---
Table C	FieldC2	String	---	---
Table E	PK_FieldE1	Number	---	---
Table E	FieldE2	String	---	---

FIGURE 1.1: Méta-modèle Weiner et al (2002)

représenter un modèle de données : noms de table, les noms des champs, types de données et les liens entre les tables. Pourtant, au nom de la simplicité ils ont sacrifié les principes de base de la modélisation, soit la normalisation des modèles. Ils ont mis toute l'information dans un seul catalogue (figure 1.1) pour sélectionner les concepts et les attributs à utiliser dans la génération de l'interface d'utilisateur.

L'exemple sert à démontrer que la méta-modélisation et la génération des prototypes peuvent être extrêmement simples. Par contre, le fait de ne pas respecter des principes de base de modélisation est une raison évidente pour ne pas poursuivre cette option.

Elsheh et Ridley[64] et Mgheder et Ridley**[114]** partagent l'approche d'utiliser les catalogues des bases de données pour générer de prototypes. Cette approche permet d'accéder directement aux données sans aucun contrôle ni configuration. Albhbah et Ridley[18, 19, 20] ajoutent un système de règles pour offrir une séparation entre les modèles des donnes et les modèles d'interface. Pourtant les règles sont limitées à la sélection des objets d'interface en utilisant la définition des catalogues de base de données. Le principal inconvénient est que le prototype construit avec cet approche est simplement une vue générique de la basse données sans aucune possibilité de définir de véritables vues d'utilisateur.

L'approche d'Antovic et al. [25] est radicalement différente ; ils s'occupent de la modélisation des cas d'utilisation et de l'interface d'utilisateur. Le modèle de données est une simple référence pour construire l'interface d'utilisateur. Ils ont bâti l'ensemble de règles sur les cas d'utilisation et on ne voit pas clairement comment ces règles sont séparées de l'interface utilisateur, ce qui ne tiendrait pas en compte le principe de séparation des préoccupations (Données, Présentations, Traitements, Gestion des évènements)[98, 89, 146]. Donc, avec l'information disponible, il s'agit d'une architecture monolithique, probablement à cause de la technologie qui les a servis d'inspiration (Ms Access). Malgré les problèmes d'architecture, ils ont vraiment exploré les détails de la spécification des cas d'utilisation. Pour le moment, la documentation des cas d'utilisation n'entre pas dans notre objectif (on vise la production de prototypes par moyen de la CGM), mais c'est un chantier important pour des futures recherches.

Forward et al. [70] ont développé un méta-langage pour la définition des modèles sous le

nom « Umple ». Ils sont capables de générer du code exécutable en autres langages (Java, PHP, C++ or Ruby). Le langage est orienté vers la définition de modèles de données et la transition des états. Pour ce qui est la définition de l'interface d'utilisateur, ils comptent sur la modification du code généré.

La proposition de Ridjanovic[139, 138] est une famille de logiciels sous le nom « Modelibra ». Modelibra concerne la définition et gestion d'une base de données «virtuelle» qui permet la création des vues dynamiques en mémoire centrale sans avoir besoin de passer par la création de tables dans une base de données. Cette fonctionnalité permet une grande flexibilité lors de la création d'un prototype, car elle évite complètement la dépendance envers les moteurs de base de données. Ainsi, l'utilisateur peut créer des enregistrements au vol pour la validation immédiate du modèle. De plus, le méta-modèle inclut la définition de la navigation entre les concepts au moyen de la définition des clés étrangères. Dans l'interface utilisateur, il y a seulement quelques possibilités de configuration, car, la personnalisation des interfaces se fait directement dans le code généré. La version actuelle de base de données « virtuelle » n'a pas une deuxième étape de persistance. Donc, à chaque réinitialisation du système on perd les données. La démarche de génération de prototypes de Modelibra concerne la génération des spécifications du modèle dans la forme des documents XML et la génération code Java pour l'exécution en utilisant le framework Apache-Wicket et des fichiers HTML et CSS.

1.3 Conclusion

Lors de cette revue de littérature, on a étudié des travaux ayant un objectif apparenté à notre question de recherche : comment la CGM peut être utilisée pour construire des prototypes des SIT. On a constaté que les objectifs et les travaux ont évolué dans la dernière décennie à partir d'un schéma limité de méta-modèle jusqu'à la construction de toute une famille d'outils pour aborder le sujet de la construction de prototypes à partir de la CGM.

On a constaté que la plupart des projets mettent l'accent sur la couche de données[158, 114, 18, 19, 20, 70, 139, 138] et que seulement Antovic et al. [25] proposent une véritable solution pour la couche d'interface. D'un autre côté, concernant l'approche de génération de prototypes, tous les travaux utilisent la génération de code et il n'y a presque pas de mention de l'option interprétation de modèles qui assure la dynamique du prototypage.

Compte tenu de ce qui précède, notre contribution originale pour répondre à la question comment la CGM peut être utilisée pour construire des prototypes des SIT réside dans la proposition d'un méta-modèle à deux couches (méta-modèle de données et méta-modèle de l'interface utilisateur) inspiré de l'approche Datarun qui intègre les caractéristiques plus importantes destinées à faciliter la communication « Client - Concepteur » et « Concepteur - Développeur ». Le méta-modèle doit considérer tous les mécanismes pour permettre l'interprétation directe des modèles.

La contribution pratique de notre recherche consiste à développer un outil qui soit capable d'interpréter la méta structure et produire un prototype fonctionnel des SIT en exploitants les concepts de la CGM. L'approche d'interprétation va être bonifiée par l'usage d'un base de données « virtuelle » mais en considérant la persistance des données tout au long de l'évolution du prototype avant la mise en œuvre de l'application finale.

Chapitre 2

Fondements

2.1 Introduction

« comme des nains assis sur les épaules de géants »
Bernard de Chartres (1115 ? - 1124 ?) [1]

La discipline de conception des systèmes d'information repose sur de nombreuses approches, méthodes ou méthodologies [2] pour obtenir un processus fiable. Les processus de construction des systèmes de information organisationnels (SIO) ont été conçus pour s'assurer que le bon logiciel peut être créé de manière revendiquée comme optimale par les auteurs de ces méthodologies. Dans ce chapitre, nous allons nous limiter à certains concepts et pratiques qui se sont révélés pertinents pour notre travail.

Dans ce travail, quand nous faisons référence à une approche, nous évoquons l'idée d'un paradigme commun qui inclut l'analyse et le développement. Nous considérons qu'il doit y avoir une continuité, si ce n'est une unicité de paradigme, quand on aborde un problème particulier. Par exemple, il est très difficile de faire une analyse structurée en même temps qu'une implémentation objet. À un moment donné, il y a un point de rupture[23] qui met en péril la réussite du projet. À notre avis une approche doit reposer sur une philosophie cohérente tout au long du projet.

2.2 Définitions

Jean-louis LeMoigne[104] modélise ainsi le système d'information au sein d'une organisation : le système informationnel assure la collecte et la mémorisation des données ou informations,

1. Jean de Salisbury, Le Metalogicon ; présentation et traduction du latin par François Lejeune, Laval, Presses de l'université de Laval ; Paris, Vrin, 2009. http ://enseignement-latin.hypotheses.org/6359

2. En pratique en SI, d'une manière que l'on peut considérer abusive, le mot méthodologie est utilisé en lieu de méthode)

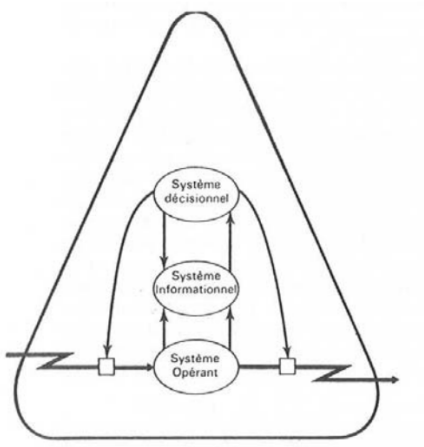


FIGURE 2.1: SO-SI-SD, la définition de Jean-Louis LeMoigne

qui résultent de l'activité du système opérant en relation avec l'environnement externe sous la direction du système décisionnel (figure 2.1).

Cette formalisation est à priori indépendante de l'usage de l'informatique, ainsi il est possible de parler du système d'information sans référence à son informatisation, en incluant des informations et processus qui échappent à toute tentative d'informatisation, dans ce sens on distingue un sous-ensemble informatisé limité à ce que l'on appelle le réel perçu qui est la partie du réel pertinente pour une organisation et qui présente un potentiel intéressant pour l'informatisation. Dans ce travail, comme nous nous limitons explicitement au domaine informatisable, nous appelons simplement système d'information organisationnel (SIO) le sous-ensemble du système d'information informatisé ou informatisable, nous l'appelons aussi, ainsi que les concepteurs de la méthode Merise, Réel perçu lorsque l'on privilégie la dimension informationnelle. Dans ce travail lorsque nous parlons du SIO, nous nous plaçons du point de vue des utilisateurs du système informatique, dans leur environnement organisationnel. Le système informatique (SI) est quant à lui réalisé par des logiciels, dans ce travail nous ne considérons que les logiciels. Le matériel étant une commodité indispensable, mais disponible, ne sera pas une contrainte dans notre contexte. Nous y incluons aussi les données, informations et connaissances créées et manipulées à l'aide des logiciels.

Il existe de nombreuses définitions des concepts "donnée", "information" et "connaissance" et parfois la même « chose » est considérée de plusieurs manières selon l'observateur et sa perspective. Dans le contexte de notre travail, et dans la perspective de la modélisation proposée par Jean-Louis LeMoigne, nous appelons donnée un élément non décomposable de représentation d'une chose ou d'un événement du réel perçu, nous appelons connaissance une abstraction permettant l'usage, l'interprétation et la manipulation des données et nous appelons information de données réorganisées à l'aide des connaissances (figure 2.2).

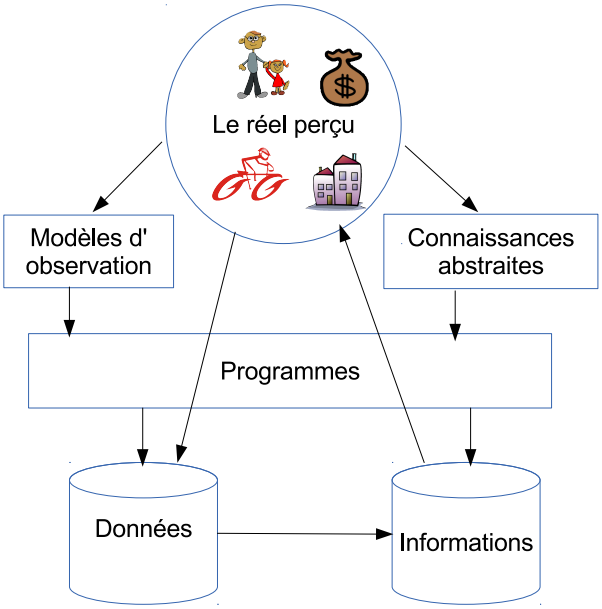


FIGURE 2.2: Définitions

Dans ce contexte, en utilisant un exemple classique :

- mon chat s'appelle mistrigri, a le poil noir et blanc et a 3 ans : sont des données, elles servent à décrire le réel perçu tel que nous le vivons concrètement, il est naturel de poser la question où est mistigri. On remarque dans cet exemple l'importance fondamentale des identifiants dans les données, ils servent à repérer une occurrence parmi toutes les occurrences.
- les chats sont des mammifères : est une connaissance, c'est une abstraction acquise ailleurs que dans notre expérience quotidienne du réel perçu, on ne se demande généralement pas où est mon mammifère, il n'y a pas d'identifiant de mammifère en tant que tel, car c'est une abstraction, une généralité. Mammifère nous apprend que l'animal a des poils, engendre des petits qu'il nourrit, a du sang chaud, etc ...
- mistrigri qui est un chat est un mammifère : est une information, elle nous permet par exemple d'inférer que mistrigi a ou aura des petits suivant que ce soit un mâle ou une femelle. Ici, l'inférence est importante, elle est le résultat de l'intégration des connaissances aux données, mais elle n'est pas totalement automatique, c'est l'usage de la banque d'information qui va faire apparaître ce que l'on recherche.

Les praticiens et les chercheurs en informatique (aussi bien sur le plan de la programmation que de l'analyse-conception) ont progressivement fait émerger trois classes de méthodes ou d'outils adaptées à chacun de ces concepts fondamentaux :

- en ce qui concerne les données en relation avec leur organisation dans des bases de données indépendantes de l'organisation des programmes, un ensemble de règles, sur

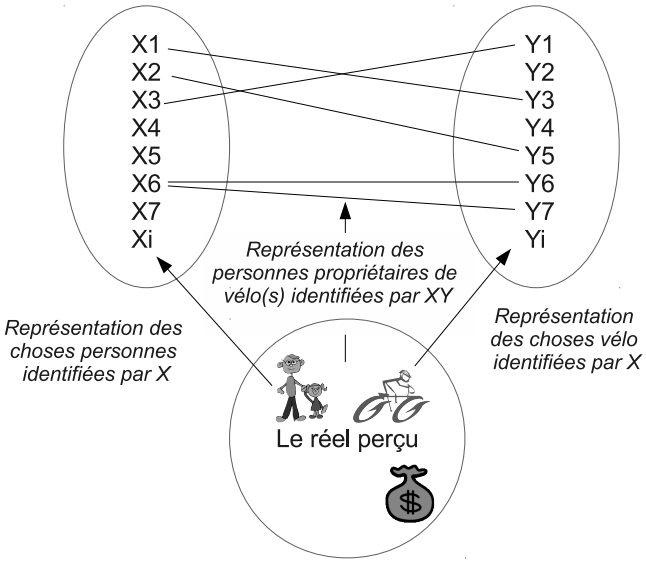


FIGURE 2.3: Fondement des modèles conceptuels de données normalisés

la base de la théorie des ensembles, est largement reconnu et adopté sous le nom de modélisation conceptuelle de données normalisées [3]. Il est résumé dans la figure 2.3 :

- en ce qui concerne les connaissances, étant donné leur caractère abstrait, il n'est pas étonnant que les outils développés l'aient été essentiellement en programmation. En effet, l'ordinateur est une machine abstraite, tout y est représenté sous la forme d'une chaîne de bits ou plus précisément de caractères (octets) : les programmes manipulent donc des « objets » abstraits que l'on peut légitimement considérer comme des connaissances. Un corpus puissant a été développé sous le nom d'approche objet et a donné lieu à des outils réputés complets et cohérents de représentation (UML) et de nombreux outils de programmation.
- en ce qui concerne les informations, notamment sur la base de la logique des prédicats [4] dans le cadre du web sémantique grâce à la puissance actuelle des ordinateurs, une approche de plus en plus explicite en ce qui concerne l'intégration des données et des connaissances pour produire d'autres informations.

Si ces trois domaines de SI peuvent aujourd'hui être présentés en tant que domaines autonomes, il n'est est pas de même dans la réalité de la programmation. En effet pour le programmeur

3. Le mot normalisé renvoie ici aux 3 formes normales des bases de données dites relationnelles
4. envisagé dès les années 70 avec le langage Prolog

les données, les connaissances ou les informations sont des variables. Ainsi les développements récents comme les langages dynamiques, les framewoks de plus en plus riches, l'architecture orientée services (SOA), les bases de données NOSQL, et plusieurs autres technologies, en se concentrant sur un domaine, concernent plus ou moins les trois domaines. Avec ce chevauchement des domaines, on a aujourd'hui, en particulier, la possibilité de réaliser des spécifications sous forme de méta données qui soient en même temps utilisées pour réaliser rapidement de manière souple et évolutive un prototype fonctionnel qui devient le socle du système en construction.

Le travail consiste dans la sélection justifiée de théorie et de concepts centrés sur des modèles issus des trois domaines d'informatisation des SI (données, connaissance et information) et leur articulation concrète autour des métadonnées avec des architectures et logiciels existants, puis la réalisation d'un outil qui permette la spécification d'un système et la génération automatique d'un prototype à partir des modèles, mais aussi à partir des mêmes métadonnées l'application complète.

La réalisation de ce travail nous a conduit à approfondir le processus de conception. Nous nous sommes appuyés sur LeMoigne[105] qui écrit :

> *« si l'on entend cet artefact que nous appelons 'modèle en cours' comme une épure sur laquelle on travaille cognitivement et on délibère (collectivement ou non), ... ce tâtonnement oscillant entre les cheminements possibles, le va-et-vient sur l'épure qui se transforme progressivement .... Jusqu'au moment qui sera plus souvent rêvé que vécu, celui qu'image ce texte émerveillant qu'est 'Eupalinos ou l'Architecte'*[5] *: « Quelle joie c'était pour mon âme de connaître cette chose si bien réglée, je ne sépare plus l'idée d'un temple de celle de son édification. En voyant cette chose, en en voyant un, je vois une action admirable » »*

Ainsi se posent plusieurs questions :

- comment et à quelle condition, les connaissances de construction du SI apportées par les framework peuvent être incorporées au SI en construction
- comment utiliser dans la construction du système de saisie des données des connaissances sur le réel perçu ou la construction des artefacts informatiques
- comment utiliser les idées informatiques sous-jacentes à la mise en œuvre de prédicats (web sémantique) pour offrir toute la flexibilité voulue à l'exploration par le prototype

2.3 La modélisation et les approches de conception

2.3.1 Sur la modélisation

5. *« EUPALINOS ou l'Architecte », Valery (1921)[154]*

« Le modèle alors, qu'il soit iconique ou symbolique,
devient source de connaissance et non plus résultat »
(LeMoigne 1987) [101]

Sur le plan théorique, Edgar Morin (1977) caractérise le problème du concepteur par le besoin d'une méthode,[6] une démarche pour permettre de concevoir en même temps l'individualité (analytique) et l'interdépendance (systémique). LeMoigne (1990) est plus prescriptif et il propose la réalisation de plusieurs modèles pour aborder la complexité : « Le passage du « Qu'est-ce concevoir ? » au « Comment concevoir ? » » va nous conduire à mettre en valeur la face instrumentale de la théorie de conception. Nous devons nous proposer quelques modèles (programmables et donc susceptibles d'être interprétés par simulation) des processus cognitifs par lesquels le concepteur-modélisateur élabore cet artefact complexe que doit être le modèle-organisation d'un phénomène présumé complexe. » Par ailleurs, [99, p307] rappelle que concevoir c'est relier et non découper comme le voudrait l'analyse classique en SIO.

Ainsi LeMoigne [101] soutient que pour faire face à un problème, nous élaborons les modèles sur lesquels nous raisonnons. Il met en lumière que pour aborder une situation complexe[7], il faut passer du registre de la recension des connaissances disciplinées à celui des méthodes d'enrichissement des connaissances actives. Il considère que la modélisation consiste à la fois à identifier et formuler quelques problèmes en construisant des modèles et en cherchant à résoudre ces problèmes en raisonnant par des simulations pour produire des modèles solutions à partir du modèle-énoncé. Ainsi, le modèle est la représentation artificielle que "l'on construit dans sa tête" et que l'on "dessine" sur un support physique. C'est un système de symboles qui agence des symboles c'est-à-dire des signes (conception technique du symbole) qui sont à la fois signifiés (la désignation : ils ont un sens pour qui les émet) et signifiants (production du sens par des symboles : ils ont un sens pour qui les reçoit,).[102, 104, 105]

Une application pratique de la démarche systémique de LeMoigne dans le champ de la conception de systèmes d'information (SIO) conduit à la formalisation d'un cycle auquel nous pouvons incorporer le concept de prototype qui est un modèle permettant la simulation : Observation, identification des frontières et processus objets du SI (le réel perçu), modélisation du SI, réalisation du prototype, validation avec l'utilisateur, répétition du cycle au besoin. Nous retenons dès maintenant le prototype, qui permet la simulation, et non celui de la maquette qui est un modèle statique qui ne permet pas la simulation nécessaire pour la créativité et la validation de la conception.

6. *« Le problème de l'observateur-concepteur nous apparaît comme capital, critique, décisif ... Il doit disposer d'une méthode qui lui permette de concevoir la multiplicité des points de vue puis de passer d'un point de vue à l'autre. Il doit disposer de concepts théoriques qui, au lieu de fermer et d'isoler les entités, lui permette de circuler productivement. Il doit concevoir en même temps l'individualité des êtres machinaux et les complexes de machines interdépendantes qui les associent ... Il a besoin aussi d'une méthode pour accéder au méta-point de vue sur les divers points de vue, y compris son propre point de vue de sujet inscrit et enraciné dans une société. Le concepteur est dans une situation paradoxale*» E. MORIN (1977, P. 179) cité par [99]

7. Selon **LeMoigne**, un système complexe est un système que l'on tient pour irréductible. Pourtant, malgré l'irréductibilité, le système est intelligible avec la modélisation.

2.3.2 Les niveaux d'abstraction (notion de couche)

L'abstraction [8] est une idée de base qui a dirigé l'évolution des approches informatique. Elle a permis aux deux volets des systèmes d'information, les données et les traitements (c'est à dire le fonctionnement de l'organisation (les processus assistés par les programmes) d'évoluer plus ou moins en parallèle. L'abstraction ainsi entendue ici est différente de celle que nous évoquions dans le premier chapitre : celle qui formalise des connaissances générales. [9]

En ce qui concerne principalement les traitements, elle prend naissance dans l'élaboration des calculs mathématiques dans un système binaire. Un peut plus tard, elle se concrétise dans les langages de programmation qui fournissent certaines fonctionnalités qui permettent au programmeur de créer des abstractions fonctionnelles, il s'agit notamment de l'utilisation des sous-routines, et des modules (programmation structurée), mais aussi des types de données.[33]

C'est cependant avec les données que la notion de niveau conceptuel a été le plus tôt et le mieux formalisée. Le niveau conceptuel se définit comme le niveau qui décrit la façon dont les utilisateurs définissent le réel perçu. Le réel perçu est la partie de la réalité concernée par le projet d'informatisation : Le modèle conceptuel des données (MCD) a pour but d'écrire de façon formelle les données qui seront utilisées par le système d'information.

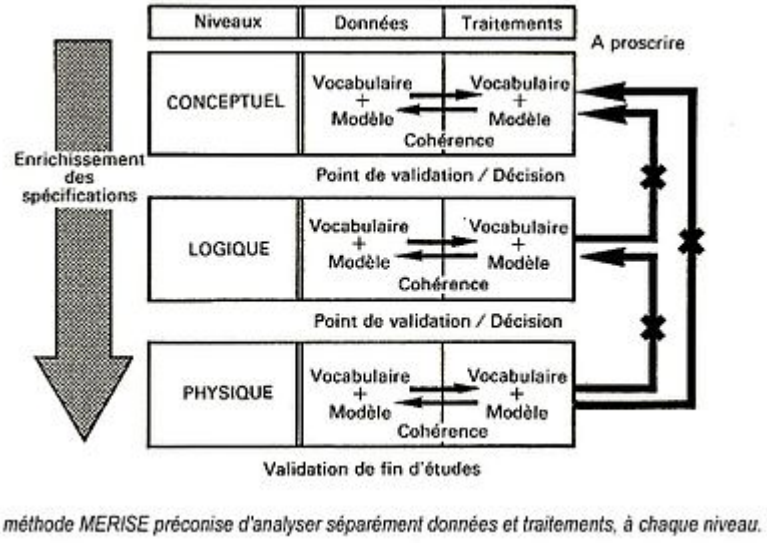


FIGURE 2.4: Les niveaux conceptuels de la méthode Merise

À début des années 70, dans le but de garantir la meilleure indépendance possible entre les données et les modules des traitements qui les exploitent, la méthode Merise propose

8. « Abstraction : Opération intellectuelle qui consiste à isoler par la pensée l'un des caractères de quelque chose et à le considérer indépendamment des autres caractères de l'objet ».[3]

9. On remarquera que la frontière entre les deux volets est floue : ainsi le respect des contraintes d'intégrité entre les données est parfois considéré comme incorporé dans les données, parfois considéré comme une composante de traitement. Cette intégration des deux volets est en pratique consacrée par l'approche objet qui associe étroitement donnée et procédure dans une même classe.

un agencement des modèles des données manipulées par les modules d'interfaces et stockées dans les bases de données. Mais l'idée est de faire explicitement une abstraction qui résulte d'une séparation en couches (figure 2.4). Cette idée d'organiser la construction de systèmes d'information en plusieurs niveaux se trouve de manière récurrente chez des nombreux auteurs, comme par exemple : Devlin et al. (1988) [56], Goikoetxea (2007) [84]. La plupart, comme dans la Méthode Merise, [61] font référence à certains ou tous les quatre niveaux : conceptuel, organisationnel, logique et physique.

Ces niveaux sont nommés conceptuel pour l'étude des fonctions et organisationnel pour l'étude de l'organisation. Le niveau définissant l'informatique est séparé en deux : un niveau décrivant l'informatique sans choix de matériel ou de logiciel précis, le niveau logique , et un niveau décrivant le résultat de la méthode ou l'informatisation finale, le niveau physique.

Dans la théorie de la programmation orientée objet, l'abstraction implique aussi la possibilité de définir des objets qui représentent des «acteurs» abstraits qui peuvent effectuer des travaux, rapporter et changer leur état, et «communiquer» avec d'autres objets dans le système. Cette abstraction est celle que nous avons évoquée dans le premier chapitre au sujet des connaissances, mais en même temps le terme encapsulation[141] se réfère à la dissimulation de la complexité interne et la définition d'une interface pour afficher l'état : ceci réfère au concept d'abstraction en couche.

L'approche objet, puis l'approche par composants permettent un niveau d'abstraction dans lequel les systèmes (données et les programmes) sont définis avec une représentation proche des besoins de l'utilisateur, tout en cachant les détails d'implémentation. En simplifiant, l'abstraction en couche peut être comprise comme la suppression des détails jugés inutiles et potentiellement différents d'une situation à une autre pour un objectif déterminé. Par exemple, dans la programmation des logiciels, des couches d'abstraction bas niveau exposent les détails du matériel informatique, où le programme est exécuté, tandis que les couches de haut niveau reflètent la logique métier. Plus on augmente le degré d'abstraction, plus la structure se rapproche de la vision de l'utilisateur.

2.3.3 L'approche structurée

Depuis l'arrivée des ordinateurs on a vu se succéder plusieurs types d'approches de réalisation des logiciels dans les systèmes d'information. Au début, l'usage des ordinateurs était constitué de calculs consistant en simples formules mathématiques. Mais au fur et a mesure que la puissance des ordinateurs a augmentée, la complexité des problèmes traités a augmentée elle aussi. En 1988, un des pionniers de la programmation, Ed Dijkstra , disait :

> "Programming started out as a craft which was practised intuitively. By 1968, it began to be generally acknowledged that the methods of program development followed so far were inadequate to face what then appeared as the so-called 'software

> crisis'. After the methods then employed had been identified as fundamentally inadequate, a style of design was developed in which the program and its correctness argument were designed hand in hand. This was a dramatic step forward."
> (Dijkstra 1988) [60]

C'était la naissance de la programmation et **l'approche structurée**. L'approche structurée considère un système à partir de la perspective des flux de données[161] et a conduit à structurer les données en fonction du traitement prévu. En ce sens si certains éléments techniques issues de l'analyse structurée (ne pas entrer à l'intérieur d'une procédure programmée) qui se retrouvent dans la conception objet sont utiles, nous ne retiendrons pas l'approche globale car elle ne s'articule pas facilement avec l'approche dite par les données.

2.3.4 l'approche par les données

> *«Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident.»* (Pike 1989)*[135]*

Suivant cette approche, l'architecture d'un système d'information doit être organisée autour d'un modèle des données mémorisées par l'organisation afin d'assurer les communications entre les acteurs de l'organisation avec le monde extérieur. Jean-Louis LeMoigne a été l'un des premiers dès 1973[100] à centrer l'architecture des SI autour des données et à ainsi poser la problématique de la méthodologie de conception des SI. En se fondant sur ce qu'il appelait le modèle des 4 flux (logistique, monétaire, personnel, actifs)[107] il proposait un renversement de perspective qui conduisait à ancrer la conception des SI dans la création (stable) des données et non dans la perspective (changeante) de leurs utilisations. Il forgeait alors le concept de générateur de données primaires.

La vision systémique[104] de LeMoigne, qui servit de berceau à la méthode Merise [10], bien que cette perspective eut souvent été ignorée par les producteurs et vendeurs de logiciel, sera par la suite à la base de la méthode Datarun qui vise à combler de manière minimaliste les lacunes de la modélisation analytique telles que la fermeture du modèle, ou le raisonnement sur un seul critère. Il part de l'hypothèse que pour aborder une situation complexe, il faut passer du registre de la recension des connaissances disciplinées à celui des méthodes d'enrichissement des connaissances actives. Selon LeMoigne le modèle est la représentation artificielle que "l'on construit dans sa tête" et que l'on "dessine" sur un support physique.

La vision systémique [103] de LeMoigne, comporte :

- Un système de règles explicites, une axiomatique, une grammaire, qui permet d'évaluer la rigueur de la construction par sa conformité à un mode d'emploi préalablement explicité et communicable.

10. En France, en particulier, la vision systémique a été explicitement un des piliers du cadre de Merise[61] puis de l'urbanisation.

– Un système de symbolisation, des paradigmes graphiques et picturaux autant que discursifs (des langages) qui permet la production de sens, l'imagination de propriétés potentielles, et par là, les représentations intentionnelles et intelligibles par et pour les acteurs des phénomènes perçus complexes au sein des quels ils interviennent, phénomènes qu'ils modélisent pour raisonner leurs conduites.

L'orientation « données » préconisée par de nombreux auteurs [129, 92, 141, 157], est basée sur la prémisse que le « modèle données » est le meilleur invariant concrètement disponible pour un système faiblement couplé. Autrement dit, « les données » sont le composante primaire, et les opérations sur les données sont subordonnées à l'existence des données.

Suivant la méthode Datarun [129] on doit réaliser le plus tôt possible le modèle conceptuel des données (MCD) car toutes les tâches du SI sont organisées autour et à partir de ce modèle de données. Ceci veut dire que la réalisation du modèle précède tout diagramme de traitement de l'information et donc du fonctionnement du SI, mais bien sûr elle implique au préalable la connaissance, et si possible la représentation, du fonctionnement de l'organisation. Les activités qui précèdent la réalisation du MCD doivent être limitées à ce qu'il est nécessaire de découvrir et de documenter pour sa réalisation. Dans la méthode Datarun les règles de gestion sont intégrées dans les modèles de données (comme l'approche objet), ainsi un seul formalisme peut suffire pour la plupart des tâches de construction du système informatique. Car, les systèmes d'information (SI) c'est de l'information et pour les utilisateurs ce sont des vues et pour le concepteur une BD normalisée. Selon la méthode Datarun il est possible de spécifier un système d'information en ces termes.

2.3.5 L'approche objet

Fondamentalement, l'approche objet, qui concerne essentiellement les représentations du monde dans les ordinateurs, conduit à penser en termes d'interaction et de hiérarchie. Alors que la complexité augmente, les éléments visibles sont les suivants :[92] [94]

1. les données qu'ils produisent et consomment,
2. les services qu'ils utilisent et offrent.

Simon (1969 [11]) [149] contribue à la théorie des objets quand il affirme qu'un système complexe est un système constitué d'un grand nombre de composants ayant beaucoup d'interactions. Pour Simon, les systèmes complexes sont composés en sous-systèmes inter-reliés qui sont composés à leur tour en leurs propres sous-systèmes. Les systèmes complexes prennent la forme d'une arborescence et il montre que l'architecte de la complexité utilise l'arborescence comme étant un schéma structurel de base. Un système hiérarchisé est le résultat de la combinaison d'un ensemble de systèmes plus simples. En conséquence, les systèmes hiérarchisés évoluent de façon plus rapide que les systèmes non hiérarchisés. Les systèmes décomposables ont des

11. La première édition de «The sciences of the artificial » date de 1969

interactions négligeables entre leurs composants internes. Aussi, il existe des systèmes quasi décomposables dont les interactions ne sont pas négligeables mais plutôt faibles. Le comportement à court terme des sous-systèmes est indépendant du comportement à court terme des autres composants, en plus le comportement de chaque sous-système à long terme n'est affecté que par l'ensemble des comportements des autres. La description des sous-systèmes et les relations entre eux permettent de rechercher et comprendre les systèmes complexes.

Suivant Simon, un objet peut être considéré comme un système, il peut représenter un concept, une idée ou toute entité du monde physique, comme une personne, un livre. Il possède une structure interne et un comportement, et il sait communiquer avec ses pairs par moyen des messages.

La modélisation objet propose donc de représenter les objets et leurs relations. L'approche orientée objet « OO » est une vue fondamentalement différente des systèmes d'information que celle observée dans l'approche structurée traditionnelle, selon LeMoigne (1994) [104, P : 193]

> « Le modèle de l'objet, quelles que soient ses propriétés, observables ou non, anticipables ou non, se construit alors autour de cet unique invariant auquel sera confiée toute description accessible du (ou des) programme(s) que le modélisateur a par ailleurs attribué à l'objet fonctionnant... Quelles que soient les transformations qu'exerce et que subit l'objet, pour qu'il demeure ce même objet il faut qu'il assure l'invariance fonctionnelle de son processeur de mémorisation. »

Les principes de l'analyse et design orienté objet (object-oriented analysis and design OOAD) sont apparus à la fin des années 80 et au début des années 90. Au milieu des années 90[121], Rumbaugh et Booch se sont associés pour combiner leurs méthodes dans un cadre unificateur qui pourrait être partagé par tous les développeurs. Plusieurs années plus tard Jacobson a rejoint l'équipe et l'UML (Unified Modeling Language) est né [12]. La version « UML version 1.0 » a été publiée en 1997 par l'OMG (Object Management Group). L'UML est au cœur de « OOAD ». UML, en tant qu'outil, prétend être indépendant des méthodologies. Quelle que soit la méthode qu'ont utilise pour effectuer l'analyse et la conception, on peut utiliser UML pour exprimer les résultats.

Selon Missikoff and Pizzicannella (1996)[115, P56] l'inconvénient majeur des méthodes « OOAD » est le manque d'une sémantique formelle. Cela conduit à des spécifications graphiques du modèle conceptuel (analyse) d'objets qui sont parfois imprécis et ambiguës. Les ambiguïtés sont souvent résolues au cours de la phase de conception ou, pire, au cours de la mise en œuvre. À partir de cette prémisse on peut conclure que le prototypage est un point intéressante a inclure dans les méthodes « OOAD » [13]. Aujourd'hui, l'orientation objet est vue davantage comme

12. Parmi les principales contributions étaient l'Object Modeling technique de James Rumbaugh, la méthode Booch de Grady Booch, et Object-Oriented Software Engineering d'Ivar Jacobson.[121]

13. «Object-oriented analysis and design (**OOAD**) is a software engineering approach to constructing software systems by building object-oriented models that abstract key aspects of the target system and by using

un paradigme. L'orientation objet est un outil efficace pour gérer la complexité des systèmes actuels.

2.3.6 L'approche par composants

L'approche par composants est un prolongement de l'orientation objet [26] [66]. Il s'agit d'une approche basée sur la réutilisation, l'exécution et la composition d'éléments indépendants faiblement couplés dans les systèmes. Au lieu de créer un exécutable monolithique, elle utilise une série de briques réutilisables. Un composant logiciel particulier est un progiciel, un site web, une ressource sur le Web, ou un module qui encapsule un ensemble de fonctions connexes (ou données).

D'autres industries ont longtemps profité de composants réutilisables. Des composants électroniques réutilisables sont utilisés dans les circuits. Presque toutes les parties dans une voiture peuvent être remplacées par un élément fabriqué à partir de l'un des nombreux manufacturiers concurrents. Des industries lucratives se sont construits autour de la construction des pièces où les interfaces standards permettent l'interchangeabilité des composants réutilisables.

L'approche par composants propose que tous les processus du système soient placés dans des composants séparés de sorte que l'ensemble des données et des fonctions à l'intérieur de chaque composant soient sémantiquement liés. En raison de ce principe, il est souvent dit que les composants sont modulaires et cohérents **[87]**.

Les composants peuvent produire ou consommer des événements, imaginons encore la voiture, pour démarrer elle a besoin d'une source de puissance, les fabricants d'une voiture moderne prennent pour acquis que c'est un composant standard avec des spécifications bien connues, tel que un voltage nominal de 12volts et un ampérage déterminé en fonction de la taille du moteur. Voilà un composant standard que n'importe quelle voiture des caractéristique similaire peut utiliser. Les fabricants de batteries n'ont presque rien à voir avec les fabricants de voitures (*le composant obéit à des standards*), mais ils travaillent ensemble. À un moment de l'histoire, le taille de la batterie est devenue aussi un standard, de façon à que physiquement on puisse changer le composant batterie pour un autre sans avoir de problème. Notre premier constat c'est que le composant batterie fournit la puissance nécessaire pour démarrer la voiture (*le composant produit quelque chose*). Pourtant la batterie n'est pas toujours en position de donner l'énergie, elle a besoin d'un message pour démarrer la voiture, c'est au moment qu'on actionne l'interrupteur qu'elle doit répondre, on dit qu'elle a reçu un message et elle répond en fournissant la puissance (*le composant perçoit des événements, les composants communiquent*

the models to guide the development process». (Rumbaugh 2003) [141] «Object-Oriented Analysis and Design (OOAD) methods are rapidly getting a wide consensus. They are mainly based on a diagrammatic approach. Diagrams are intuitive, fast to learn and easy to use. However, they lack of a formal basis and their semantics is mainly descriptive. Furthermore, validation and verification of analysis specifications are, to a large extent, performed manually.» (Missikoff and Pizzicannella 1996:56)[115]

par messages). Mais, allons plus loin, la batterie ne peut pas fournir de l'énergie de façon indéfinie, elle doit être rechargée, quand la voiture est en fonctionnement elle va consommer de l'énergie pour être capable de garder son niveau de charge (*le composant consomme ce dont il a besoin*). De plus, elle est capable de communiquer certains aspect de son état, si la charge est faible elle peut faire allumer un indicateur (*Le composant produit des événements pour communiquer son état*). Un dernier point très important c'est que le fabriquant d'automobiles n'a pas besoin de connaître les techniques de production ni de fonctionnement des batteries (*Le composant cache sa complexité*). D'un autre côté, la batterie peut être utilisable comme un source de puissance pour démarrer un bateau, ou pour alimenter n'importe quelle autre machine (*Le composant est isolé de son environnement externe*).

L'approche composants est considérée comme la base de l'orientation service et les architectures orientées services (SOA). Dans ce contexte un composant est converti par les services Web (SOAP, REST) dans un service accessible par internet et hérite ensuite des nouvelles caractéristiques au-delà de celle d'un composant ordinaire.

L'approche composant peut être vue comme une forme aboutie de l'orientation par objets, ou plus simplement comme un assemblage de briques logicielles prédéfinies, et conçues dans le but d'être réutilisées (ce qui n'est pas forcément le cas d'objets définis dans un contexte particulier et réutilisés par hasard). Les frameworks développés avec la philosophie du logiciel libre fournissent aujourd'hui des ensembles de composants que l'on on peut librement utiliser. Mieux, avec les licences libres, on peut aller chercher un logiciel, ouvrir le capot (comme dans un voiture), vérifier comment et quels composants elle utilise puis extraire les composants que sont importants pour les réutiliser dans un nouveau contexte.

En conclusion, on peut dire que si la crise du logiciel est loin d'être terminée, elle est mieux connue, et que les avancées rapides de la technologie rendent possibles des applications jusqu'alors inenvisageables et obligent sans cesser à reconsidérer des « vérités » que l'on croyait acquises.

L'informatique est aujourd'hui une industrie mure, et il existe différentes approches selon le type de besoin et le contexte dans lequel le produit sera utilisé. Les contraintes seront différentes selon que l'on fabrique un logiciel pour une garder un liste de chansons ou un logiciel stratégique et vital pour une population.

L'approche objet et ses corollaires (composantes, services, agents) est parvenue à un point de maturité tel qu'elle peut répondre aux exigences de décomposition hiérarchiques exposées par Simon (1969) dans son analyse de la complexité[149].

Le principe de base [143] [54]« tout est un objet » peut continuer à évoluer, mais la théorie qui a donné vie aux objets est encore actuelle et on ne trouve pas encore dans la littérature spécialisée un théorie radicalement nouvelle. L'informatique est une science en permanente évolution, il faut trouver un autre chemin pour continuer son développement.

2.3.7 La réalisation à partir de modèles

Étant donné que les composantes logiciel sont arrivées à une certaine maturité, le niveau suivant vise à élever le niveau d'abstraction dans la spécification des programmes et d'accroître l'automatisation dans l'élaboration des programmes.

L'idée de la génération à partir des modèles n'est pas nouvelle, elle remonte aux années 80[68], elle était considérée comme le « Graal » de la conception des SIO. Le premier élément abordé était l'automatisation de l'interface utilisateur par moyen de la génération de code.

En novembre 2000, l'OMG a rendu public l'initiative MDA[32]. L'objectif de cette initiative est fondé sur les avantages qui pourraient être tirées d'un passage de la réalisation ad-hoc du code à des pratiques fondées sur des modèles pour d'une façon directe ou indirecte utiliser des composants réutilisables.

L'idée promue par MDA consiste à utiliser des modèles pour le développement de systèmes, augmentant ainsi le niveau d'abstraction dans la spécification des programmes. Les modèles de niveau supérieur y sont transformés en modèles de niveau inférieur jusqu'à ce que le modèle puisse être rendu exécutable en utilisant soit la génération de code soit de l'interprétation du modèle[37].

Le principe *« Tout est objet »* a été très utile dans la conduite de la technologie dans le sens de la simplicité, de la généralité et la puissance de l'intégration. De même, en MDA, le principe fondamental selon lequel *« Tout est un modèle »* possède de nombreuses propriétés intéressantes.

L'idée d'un modèle exécutable permet de concevoir le SIO directement en fonction des concepts du domaine (réel perçu). Cela permet aux experts du domaine de comprendre la représentation du système informatisé. Ainsi, des experts du domaine peuvent être en mesure non seulement de valider la satisfaction de leurs besoins, mais ils peuvent aussi apprendre à modéliser le système eux-mêmes. Dans un cas idéal, un système doit être modélisé directement par les experts du domaine qui en ont besoin.

Dans cette approche, l'étape suivant la modélisation est la construction d'un ensemble de règles (conception déclarative) dans lequel une description précise des propriétés est suffisante pour contrôler les aspects les plus importants d'un SIO. Autrement dit, on peut passer d'un modèle à un meta-programme pour spécifier le SIO.

Pourtant, il a ses écueils dans la pratique. Par exemple :

- Les langages de déclaration ne sont pas suffisamment expressif pour faire tout ce qu'il faut dans un SIO.
- Les frameworks rendent normalement très difficile d'étendre le logiciel au-delà de la partie automatisée.

– Les détails que sont ajoutés après la génération du code, sont faciles a perdre s'ils ne sont pas remontés au modèle.

Nous pensons que tôt ou tard, l'évolution des outils va permettre de palier ces inconvénients. Actuellement c'est la meilleure approche pour la génération des prototypes.

2.4 L'Architecture

> *« The software industry has a delight in taking words and stretching them into a myriad of subtly contradictory meanings. One of the biggest sufferers is "architecture". I tend to look at architecture as one of those impressive sounding words, used primarily to indicate that we are talking about is something that's important. »*
>
> (Fowler 2003) [72]

Tel qu'énoncé par Fowler, il y plusieurs conception d'architecture en ce qui concerne les organisations, mais à la base c'est une métaphore pour aborder la conception et la construction d'un système complexe. C'est l'abstraction en couches (imbrication) ou en composants (blocs) pour aborder la complexité.

Pensez à la fabrication de voitures [14] comme une métaphore. Les travailleurs impliqués dans la fabrication d'automobiles peuvent se spécialiser dans la production de pièces, mais, ils ont souvent une vision limitée de l'ensemble du processus. Ils considèrent la voiture comme une énorme collection de pièces à assembler. Une voiture est beaucoup plus que cela. Une bonne voiture commence par une vision, puis les spécifications soigneusement écrites, continue avec le design, beaucoup de conception, sûrement du prototypage. Enfin, après de nombreuses modifications et raffinages, le produit reflète la vision originale. La conception ne se fait seulement sur le papier. Une grande partie consiste à faire des modèles et des prototypes, les tester sous diverses conditions pour voir si elles fonctionnent. La conception est modifiée en fonction des résultats d'essais.

Le développement du logiciel est similaire. Nous ne pouvons pas simplement nous asseoir et taper le code définitif. Cette méthode ne peut fonctionner que pour les situations triviales. Mais nous ne savons pas créer un logiciel complexe d'un seul coup. Est-il possible de créer des logiciels bancaires complexe sans une bonne connaissance du domaine ? Impossible[65]. Une système quel qu'il soit est très bien compris par les personnes à l'intérieur, par leurs spécialistes, ils connaissent tous les détails, toutes les créations de données, toutes les questions possibles, toutes les règles. Mais, elles ne savent pas comment construire des systèmes d'information. On a besoin de transfert de connaissances et de travail en équipe.

14. cette métaphore est utilisé par plusieurs auteurs entre autres Eric Evans.

2.4.1 Artificiel : Artefacts

Le mot artificiel est défini dans le sens « fait par l'homme, par opposition à naturel, fait par la nature ». Simon[149] définit les artefacts comme « une interface entre un environnement interne, un environnement externe et l'organisation de l'artefact lui-même ». Il ajoute qu'un artefact peut atteindre ses buts lorsque l'environnement interne s'adapte à l'environnement externe. En autres mots, Simon énonce que l'objectif essentiel des activités de l'ingénierie c'est la construction des artefacts (des objets artificiels), pour lier un environnement externe (la réalité) au un environnement interne (l'artefact), et que les principaux avantages d'une distinction entre les environnements interne et externe sont :

- pouvoir prédire le comportement du système adaptatif ou artificiel connaissant son environnement externe, ses objectifs, et quelques hypothèses sur son environnement interne,
- séparer le système interne de son environnement afin de garder une relation indépendante entre ce dernier et ses buts.

Un SIO est un élément artificiel de l'environnement. Donc, il est indispensable d'adopter une méthode de conception. Les SIO intègrent plusieurs artefacts qui assurent les fonctions de : mémorisation des informations génération des informations, diffusion des informations.

2.4.2 Zachman

Le mot architecture a été popularisé par Zachman dans une métaphore[163] qui compare la construction d'un système informatique à la construction d'une maison ou d'un avion. Il a ainsi montré qu'il n'y avait pas une seule représentation de la construction d'un produit complexe, mais qu'il y avait différents intervenants avec chacun ses propres points de vue sur le projet. Pour lui, ce sont des « Perspectives» qu'il nomme, suivant les pratiques du domaine, [134] : contextuelle, conceptuelle, logique, physique, détaillée. Zachman a aussi combiné les perspectives avec les questionnements de base de la communication humaine, et qu'il a appelée « Dimensions »[164] : « quoi » (données : que traite-t-on ?), « comment » (fonctions : comment traite-t-on ?), « qui » (personnes : qui est impliqué ?), « ou » (réseau, organisation : où traite-t-on ?), « quand » (temps : quand traite-t-on ?, selon quels cycles ?), « pourquoi » (objectifs, contraintes : pourquoi veut-on traiter ?) [15]. Ces mêmes questionnements ne sont pas surprenants, ils ont été utilisés par les philosophes de l'antiquité [16][**?**] pour comprendre le monde et c'est aussi la base de l'ontologie « Qui appartient à la catégorie de l'être et non à celle du paraître » [8].

> «The Zachman Framework™ is an ontology - a theory of the existence of a structured set of essential components of an object for which explicit expressions is necessary and perhaps even mandatory for creating, operating, and changing

15. dimensions que l'on trouvait déjà, par exemple dans Merise soit explicitement, les données, traitement et communication, soit implicitement dispersées dans les précédentes, voir par exemple Tabourier, de l'autre côté de Merise).
16. [16]

the object (the object being an Enterprise, a department, a value chain, a "sliver," a solution, a project, an airplane, a building, a product, a profession or whatever or whatever).»
Zachman Interantional Enterprise Architecture[90]

Les **perspectives** sont abordées selon les points de vue des différents acteurs impliqués (l'équipe) : le manager qui arrête les plans et la stratégie de l'organisation, les métiers qui agissent dans l'organisation, l'analyste fonctionnel qui modélise les processus métier, le concepteur qui choisit les technologies susceptibles de répondre aux enjeux métier, le développeur (sous-traitant) en charge du développement des applications, les utilisateurs qui doivent en bénéficier. Le résultat c'est une matrice de 36 cellules qui couvrent les différentes problématiques de l'entreprise (tableau 2.1). Chaque cellule peut avoir un ou plusieurs éléments importants dans un projet, et chacun de ces éléments reçoit le nom d'artefact (voir 2.4.1).

The Zachman framework

Primarily responsible		Data (what)	Function (how)	Network (where)	People (who)	Time (when)	Motivation (why)
Executive management overview	Objectives/ scope	List of things important to the enterprise	List of processes the business performs	List of locations where the enterprise operates	List of organizational units	List of business events/cycles	List of business goals/strategies
Senior operating management	Enterprise model	Entity relationship diagram	Business process model (physical data flow diagram)	Network configuration (nodes and links)	Organization chart (roles; skill sets and security needs)	Business master schedule	Business plan
Business system architect	Information system model	Data model (converged entities, fully normalized)	System data flow diagram; Application architecture	Business system architecture	Human interfaces (roles, data access rights)	Dependency diagram, entity history (process structure)	Business rules model
Information system architect	Technology model	Data architecture (mapped to linked systems)	system design; structure chart, pseudo-code	Technology system architecture	User interface (what the user will see); security design	Control structure diagram	Business rules design
System analyst/ programmer	Detailed system definition	Data design (denormalized); physical storage design	Detailed program design	Systems and network architecture	Screens and security architecture (who sees what)	Timing definitions	Rule specification in program logic
User and service provider	Implemented and operating system	Converted, initialized and production data	Conversion and production application programs	Systems and network infrastructure	Trained user, operating and support personnel	Real business events/ activities	User procedures and system enforced rules

TABLE 2.1: The Zachamn framework [90]

En résume, la conception du logiciel est comme la création de l'architecture d'une maison, il s'agit d'un grand projet partagé par plusieurs personnes. L'architecture [84] est un ensemble d'artefacts d'affaires et d'ingénierie, y compris la documentation textuelle et graphique qui décrivent et guident le fonctionnement d'un SIO, y compris les instructions de fonctionnement de son cycle de vie, de sa gestion, et de son évolution et de son maintenance. La structure (Framework) de Zachman n'est ni une méthodologie ni un méthode, mais elle est définie comme une ontologie pour décrire l'entreprise (pour organiser les artefacts), c'est un outil de cadrage qui répond à différentes questions et qui évolue avec le temps et les technologies[162]. Le prototypeur, comme nous le proposons, est un excellent outil pour stocker les différents artefacts de l'architecture d'entreprise.

2.4.3 Différentes sortes d'architecture

> «When discussing architecture, it is important to define the scope: Are we taking an organization-wide view, or are we talking about one information system? Hence a common distinction in the past was between an enterprise-wide architecture and an information system's architecture (sometimes called software architecture). While the latter is limited to the elements of just one system, the former represents a framework for all information systems in the organization. » (Kurbel 2008)[97]

Les systèmes d'information comme toute autre système sont des collections de composantes qui une fois mis ensemble, suivant un ensemble de principes, forment une unité. Unité qui peut à son tour être un élément d'une collection. La structure d'un système informatique dépend du point de vue adopté (perspective) et des éléments met en évidence. Il peut par conséquent y avoir plusieurs diagrammes d'architecture pour un même système.

> « During the last decade, enterprise architecture has grown into an established approach for holistic management of the information systems in an organization »(Johnson et al 2006) [91]

L'architecture d'entreprise est la logique structurante pour les processus métiers et l'infrastructure informatique, reflétant les exigences d'intégration et de standardisation du modèle opératoire de l'entreprise. L'architecture d'entreprise fournit une vision à long terme des processus, des systèmes et des technologies de l'entreprise afin que les projets individuels puissent construire des capacités et non pas simplement répondre à des besoins immédiats.

> « L'urbanisme des systèmes d'information est un moyen pour sauvegarder la cohérence et améliorer l'efficacité du système d'information c'est à dire la qualité de sa contribution à l'atteinte des objectifs de l'entreprise » (Longépé, 2001).

Parallèlement en France, l'urbanisme[112] vise à créer une rupture dans la façon d'appréhender le système d'information. En considérant les logiciels comme des éléments inter reliés dans un ensemble général, l'urbanisme a pour but de définir les principes de construction et de connexion de ces éléments pour assurer à l'ensemble la souplesse d'adaptation et l'évolution nécessaire. En ne se focalisant plus seulement sur les actions ponctuelles mais en portant un regard global sur le système d'information et en le considérant comme un tout, il se donne pour objectif d'assurer sa cohérence et sa performance et de lui donner une stratégie propre, lui permettant d'accompagner l'entreprise face à ses nécessités [57].

La démarche d'architecture TOGAF[85], considère (figure 2.5) :

- A. Vision d'architecture
- B. Architecture d'affaires
- C. Architecture de systèmes d'information et
- D. Architecture technique

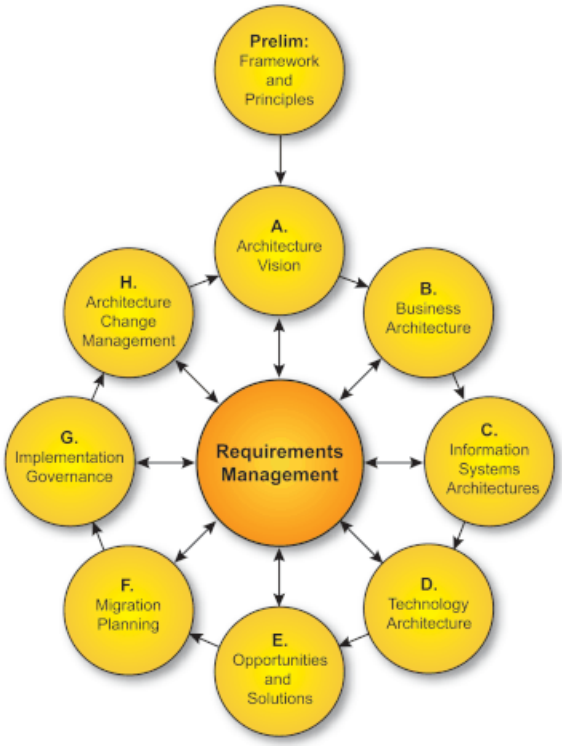


FIGURE 2.5: Togaf - Architecture Development Cycle (ADM) [6]

Il y d'autres classifications d'architecture, mais nous nous arrêtons sur TOGAF, notamment nous visons l'architecture d'affaires (B) pour ce qui concerne la démarche de conception et l'architecture de systèmes d'information (C) , pour ce qui est le développement de notre outil de prototypage.

2.5 Cycles Vie

> « The techniques that have been developed for automatic programming over the past five years have mostly aimed at simplifying the part of programming .... As a result of progress in this area (and a growing number of experienced programmers), we find that large programs can now be produced; unfortunately, they are difficult to test and document. If the newest very-high-speed, large-memory computers are to be fully utilized, we must develop automatic programming procedures so that they allow cheap production of highly reliable, easily revised, well-documented system programs. »
>
> Benington (1956) [33]

Pour la plus part le développement de logiciels est une activité chaotique[73], souvent caractérisée par la pratique "code and fix" . Le logiciel est écrit sans avoir un plan sous-jacent et la conception du système est bricolée à partir de nombreuses décisions à court terme. Cela fonctionne effectivement très bien quand le système est petit, mais quand le système se développe, il devient de plus en plus difficile d'ajouter de nouvelles fonctionnalités au système. En outre les bogues deviennent de plus en plus nombreuses et de plus en plus difficiles à corriger.

L'informatique est une discipline très récente en comparaison aux autres disciplines comme la construction de bâtiments ou plus récemment la construction des appareils mécaniques comme les voitures ou les avions. La tendance naturelle était des emprunter des méthodes des ces disciplines pour assimiler la construction de systèmes d'information à procédures de construction déjà connues[163]. Le mouvement original pour essayer de rendre le développement logiciel plus prévisible et plus efficace à commencé avec l'introduction de la notion de méthodologie. Un méthodologie impose un processus rigoureux et détermine un cycle de vie. En informatique (comme dans autres disciplines), un cycle de vie couvre toutes les étapes du création de logiciel depuis sa création à la définition des besoins jusqu'à mise en service et la maintenance.

2.5.1 Les méthodes séquentielles (cascade)

Une des méthodologies [17] de développement de produits logiciels qui était populaire dans les années 70 et 80 est la méthode séquentielle également connue sous le nom de modèle « waterfall » ou cascade. La programmation des SI à cette époque était un processus analogue à un processus mécanique, il a donc développé un processus avec une forte orientation sur la planification inspirée par d'autres disciplines de l'ingénierie.

Le modèle en cascade de Benington (1956)[33] (figure 2.6) a servi de base a tous les autres modèles[142], car il a définit les phases nécessaires et a postulé les étapes de recueil de besoins et d'analyse précédant la phase de codage. Royce en 1970[140] a amélioré le modèle original en fournissant des boucles de rétroaction de telle sorte que chacune des étapes précédentes peut être réexaminée (figure 2.7). L'approche a bien fonctionné dans des environnements stables, mais son efficacité a été mise en cause dans des environnements incertains et dynamiques. Car, l'approche séquentielle peut nécessiter de très longues périodes de temps pour obtenir des résultats. Les gestionnaires peuvent également éprouver des difficultés à évaluer les progrès réalisés précisément parce que les tests se trouvent trop tard dans le cycle de développement.

Autres modèles moins connues ont était dérivés du modèle en cascade, Ruparelia[142] documente le modèles : B, V, B+, entre autres. Mais, ils ne sont pas arrivés à fournir un solution à la « crise de logiciel »

17. Il est nécessaire de garder à l'esprit que le modèle est différent d'une méthodologie dans le sens que le premier décrit ce qu'il faut faire alors que le second, en outre, décrit comment le faire. Ainsi, un modèle est descriptive, tandis qu'une méthodologie est prescriptive.

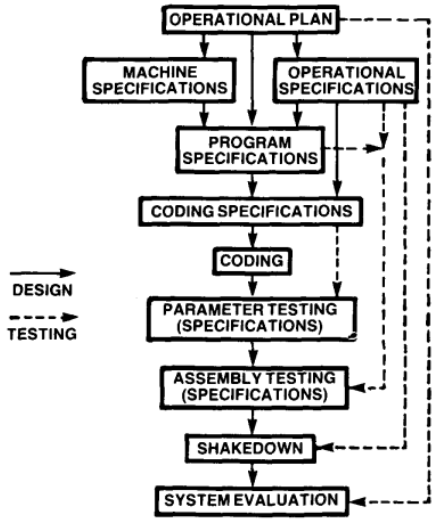


FIGURE 2.6: Original « Program production » schema de Benington [33]

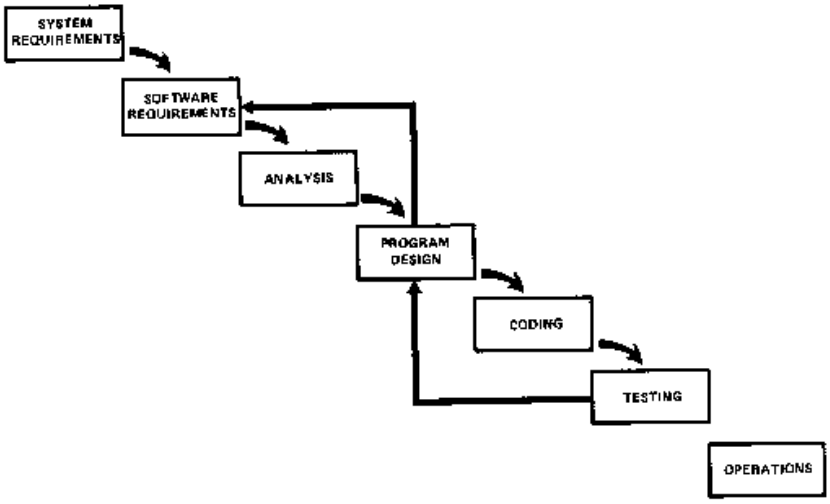


FIGURE 2.7: Méthode de cascade avec boucles de rétroaction [140]

Le défi pour la construction des systèmes d'information dans des environnements incertains et dynamique, c'est que les besoins des utilisateurs pour de nombreux types de logiciels sont si difficiles à comprendre qu'il est presque impossible ou imprudent d'essayer de concevoir le système complètement à l'avance, d'autant que l'environnent et les besoins (désirs) des clients sont constamment en changement et qui évoluent rapidement. Donc, les résultats n'étaient pas très bons et les critiques sur la méthode étaient très fortes.

> «The "waterfall model" may be unrealistic, and dangerous to the primary objectives of any software project»
> (Gilb 1985) [83]

Alors, comment peut-on contrôler dans un monde imprévisible ? La partie la plus importante, et encore difficile est de connaître exactement la situation. On a besoin d'un mécanisme de rétroaction[73] intègre qui peut dire exactement quelle est la situation à intervalles réguliers. La clé de cette rétroaction est le développement itératif.

2.5.2 Les méthodes itératives

Une panoplie des méthodes ont était élaborées en réaction aux méthodes séquentielles. Dans l'approche itérative, le cycle de développement de produits est divisé en sous-cycles, chaque sous-cycle étant constitué de une mini cascade : concevoir, développer, construire, essayer. Le modèle en spirale de Boehm[41] mérite une mention spéciale, il modifie le modèle de cascade en introduisant plusieurs itérations (qu'il a appelé spirales) à partir de petits commencements. La philosophie de la méthode en spirale consiste à commencer petit pour penser grand. À chaque cycle dans le progrès de la spirale, un prototype est construit, vérifié par rapport à ses besoins et validé par des tests. Boehm (1988), décrit le modèle en spirale, comme une innovation qui permet la combinaison de la gestion conventionnelle, le prototypage et les modèles incrémentaux. En général les méthodes itératives mettent l'accent sur la capacité de répondre à des nouvelles informations de l'environnent (client) en utilisant des techniques de rétroaction au cours d'un cycle de développement.

2.5.3 Manifeste Agile

Un autre variation des méthodes itératives, sont les méthodes dites « légères ». Ces méthodes tentent un compromis utile entre l'absence de processus et trop de processus[73], fournissant juste assez de processus pour obtenir un gain raisonnable. A l'époque (avant 2001), il n'y avait aucun nom pour nommer ce groupe des technologies, le surnom de «léger» avait grandi autour d'eux. Mais, la plupart des personnes impliquées pensaient pas que c'était le bon terme car il ne rendait pas compte de la spécificité de ces approches[73].

Le terme « Agile » est apparu au début de 2001 quand un groupe de personnes qui avaient été fortement impliquées dans les travail avec des méthodes « légères » se sont réunis pour échanger des idées que finalement est abouti avec le « Manifeste Agile ». Agile devint une méthodologie, une façon de faire les choses, qui encadre plusieurs méthodes qui respectant les principes énumérés dans le « Manifesto »(Beck et al. 2001) [31].

Les méthodes Agile sont souvent identifie avec « Scrum » (figure 2.8), qui est « une des méthodes » agile dédiée à la gestion de projets.

> « Scrum is a simple framework for effective team collaboration on complex projects. Scrum provides a small set of rules that create just enough structure for teams to be able to focus their innovation on solving what might otherwise be an insurmountable challenge. »[12]

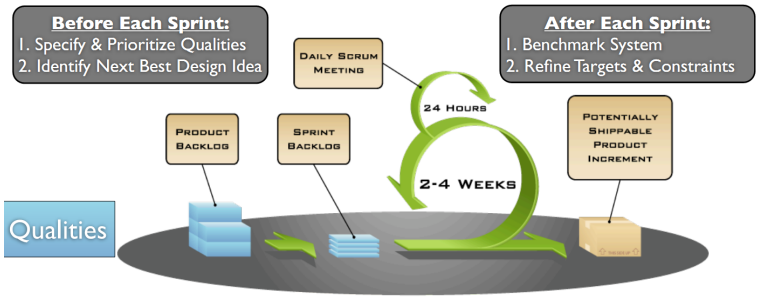


FIGURE 2.8: La méthode Scrum[12]

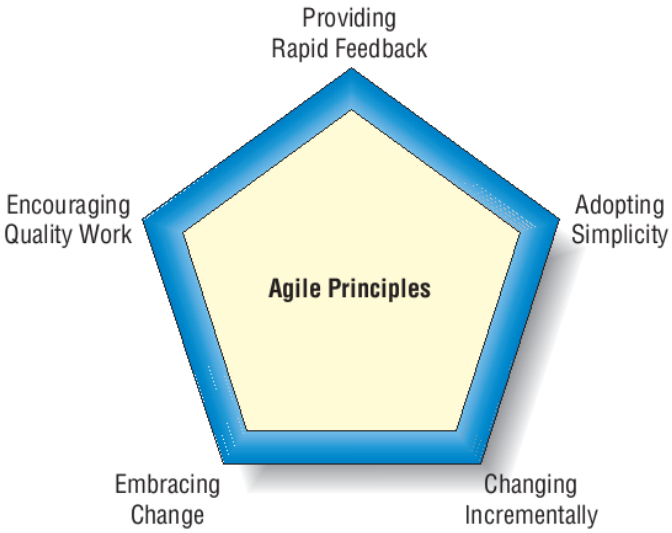


FIGURE 2.9: Principes AGILE [93]

Le terme «agile» se réfère[73] à une philosophie de développement de logiciels. Ce large parapluie abrite de nombreuses approches plus spécifiques telles que l'Extreme Programming (XP), Scrum, Lean développement, Dynamic Systems Development Method (DSDM), Crystal, (Rational) Unified Process, etc. Chacune de ces approches a ses particularités, ses communautés et ses promoteurs. La plupart, mais pas tous, de ces méthodes étaient sortis de la communauté du logiciel orienté objet qui a longtemps préconisé des approches itératives de développement.

Chaque communauté est un groupe distinct, mais pour être catalogué Agile, il doit suivre les mêmes principes généraux[73] (figure 2.9) . Chaque communauté emprunte ainsi des idées et des techniques de l'autre. De nombreux praticiens fréquentent différentes communautés et la propagation des idées différentes autour et dans l'ensemble du écosystème « Agile » est complexe et dynamique. Le résultat de tout cela est que les méthodes agiles ont apporté des changements importants sur les méthodes basées concernant l'ingénierie du logiciel. La différence la plus immédiate, c'est qu'ils sont moins utilisateurs de documents formels, n'exigeant généralement qu'une petite quantité de documentation pour une tâche donnée. En résume [147]

	Traditional	Agile
Fundamental Assumptions	Systems are fully specifiable, predictable, and can be built through meticulous and extensive planning.	High-quality, adaptive software can be developed by small teams using the principles of continuous design improvement and testing based on rapid feedback and change.
Control	Process centric	People centric
Management Style	Command-and-control	Leadership-and-collaboration
Knowledge Management	Explicit	Tacit
Role Assignment	Individual—favors specialization	Self-organizing teams—encourages role interchangeability
Communication	Formal	Informal
Customer's Role	Important	Critical
Project Cycle	Guided by tasks or activities	Guided by product features
Development Model	Life cycle model (Waterfall, Spiral, or some variation)	The evolutionary-delivery model
Desired Organizational Form/Structure	Mechanistic (bureaucratic with high formalization)	Organic (flexible and participative encouraging cooperative social action)
Technology	No restriction	Favors object-oriented technology

TABLE 2.2: Comparaison Cascade - Agile [120]

[82] « Agile » ne s'agit pas simplement de produire du code, mais il s'agit de livrer de la valeur pour les parties prenantes. « Agile » n'est pas centrée sur le travail de programmation, il s'agit de faire travailler des systèmes, pour des personnes ayant des besoins réels.

Le tableau 2.2 résume les différentes caractéristiques de l'approches agile vs l'approche en cascade.

Les principes de méthodologies agiles sont analogues aux idées délimitées en méthodologie des systèmes souples de Checkland (SSM Soft system Metodologie)[47]. Ceux-ci reflètent les caractéristiques essentielles des systèmes complexes adaptatifs, et ont le potentiel de doter les organisations et les systèmes de propriétés émergente[120].

Checkland recommande de ne pas passer beaucoup de temps dans la construction du modèle initial. Il estime qu'il est préférable de procéder à l'étape de comparaison, avoir des discussions, le gain idées, et revenir au modèle, plutôt que de passer beaucoup de temps sur la construction du modèle initial. Cela renforce sa conviction que le processus de SSM de même que les processus AGILE sont basé sur la communication, le débat et l'apprentissage plutôt que la production de la solution «idéale» à la première fois.

2.5.4 La cathédrale et le Bazar

> "I'm basically a very lazy person who likes to get credit for things other people actually do."
> Linux Torval [137, P6]

Le modèle de développement de logiciels open source (également connu sous le nom « cathédrale et le Bazar »[137]), présente plusieurs similitudes avec les méthodes « Agile » [73] [52]. notamment, la plupart des projets open source partagent des principes avec le Manifeste Agile :

- favorable aux changements (« *The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is bette*r »),
- participation des utilisateurs (« *Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging* »),
- rétroaction continue, livraison continue de fonctionnalismes réelles («*Release Early, Release Often. And listen to your customers* »),
- respect des collaborateurs et des utilisateurs («*If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource* »).

À notre avis, la méthode de développement communautaire (cathédrale et le Bazar) est une forme spéciale de développement Agile dans laquelle la collaboration et la communication sont à la base d'une action collective. Diverses parties prenantes, y compris les développeurs et les utilisateurs finaux passent par des cycles répétés de « *faire pour comprendre et de comprendre pour faire* »[106] qui favorisent un environnement d'apprentissage et d'adaptation. Les membres de l'équipe sont dotés de pouvoirs et ne sont pas confinés à un rôle spécialisé. Cela augmente la diversité et la variété des équipes et leur permet de s'auto-organiser et répondre avec empressement aux situations émergentes.

2.6 Logiciel Libre

> « *When I think of the computer programs I require daily to get my own work done, I cannot help but realize that none of them would exist today if software patents had been prevalent in the 1960s and 1970s* »
>
> Donald Knuth [18]

Le logiciel libre apparaît dans ce chapitre pour au moins deux raisons, sans l'usage de ces logiciels notre projet n'aurait pas pu être mené à bon terme, mais aussi et surtout, car nous ne considérons pas le logiciel libre comme une technologie particulière, mais une pratique de la programmation tirant parti du travail en commun.

Le logiciel est un objet complexe qui nécessite de très nombreuses composantes qui évoluent continuellement en fonction des opportunités et des besoins. Ceci repose sur de la créativité et il n'est pas question de réinventer la roue pour chaque nouveau développement. Les premiers logiciels au début des années 50 étaient développés et partagés librement par les

18. Dans une lettre adressée au Patent Office [95]

programmeurs[159], la question de leur propriété et de leur licence ne se posait alors pas, puis avec l'essor des ordinateurs et l'indépendance progressive du logiciel par rapport au matériel, des entreprises productrices du logiciel ont établi leur propriété sur le code qu'elles produisaient et l'ont protégé par le droit d'auteur puis dans les pays qui l'acceptait, les brevets. Cependant, depuis les années 80 essentiellement sous l'impulsion initiale de Richard Stallman la liberté d'utiliser le logiciel a été formalisée et rendue pérenne grâce à la création d'une licence, la licence GPL.

Richard Stallman est motivé par la nécessité de protéger les libertés d'usage d'un logiciel, qu'il considère comme garantes de la liberté individuelle, et s'appuie sur la notion de bien commun. La licence qu'il a créée en compagnie de Eben Moglen[117] garantit pour tous et pour quelque usage que ce soit :

- la liberté d'étudier le fonctionnement du programme, et de l'adapter à ses propres besoins ;
- la liberté de redistribuer des copies de façon à pouvoir aider son voisin ;
- la liberté d'améliorer le programme, et de diffuser ses améliorations au public, de façon à ce que l'ensemble de la communauté en tire avantage ;
- la liberté d'améliorer le programme, et de diffuser ses améliorations au public, de façon à ce que l'ensemble de la communauté en tire avantage.

Le mouvement open source a pris de l'ampleur, non seulement pour des raisons morales (qui sous-tendent l'action de Richard Stallman qui qualifie les logiciels sous licence fermée de mauvais), mais aussi pour des raisons d'efficacité technique et économiques. Le partage du logiciel sous licence libre est reconnu comme un moyen très efficace de qualité et pour l'innovation de manière économique. C'est ainsi que la philosophie du logiciel libre a été adopté par de nombreuses entreprises et gouvernements et que des logiciels complexes comme ceux de Facebook[14] ou de Google[55] ont été construits essentiellement avec des logiciels *libres*. Ainsi Donald Knuth dit :

> « The success of open source code is perhaps the only thing in the computer field that hasn't surprised me during the past several decades. But it still hasn't reached its full potential; I believe that open-source programs will begin to be completely dominant as the economy moves more and more from products towards services, and as more and more volunteers arise to improve the code. » (Knuth et Binstock 2008) [96]

L'open source est devenu un réseau de coopération et connaissance distribué et fait que les outils informatiques (développement et exploration) ont changé radicalement et rapidement constituant ainsi un patrimoine de logiciel dans lequel on peut librement puiser. Ainsi choisir une stratégie reposant sur du logiciel libre « is not only a choice of software, but also a means of acquiring knowledge » (Gilberto et Fonseca, 2007) [19]. Il existe de nombreux projets de logiciels

19. Cité par Pascot [131]

libres qui peuvent être une source inestimable d'apprentissage, par exemple, récemment (avril 2013), le site de répertoires public GitHub (ouvert en 2008) répertorie 6 000 000 de projets sous la rubrique « logiciel libre »[1] et 3 500 000 contributeurs. Il n'existe plus une seule compagnie qui puisse atteindre ces chiffres et cette capacité d'innovation. De nos jours un grande quantité des projets informatiques se font à partir de logiciel libre :

> *« Most modern and complex services and products are run through software. For example, airplanes and track control systems are run by very complex and safety-critical software. Millions of lives are everyday under the responsibility of these systems »* (Fuggetta, 2003)[78].

Avec l'accès à l'immense patrimoine numérique fournit par le logiciel libre, on peut réaliser aujourd'hui de grands projets avec un moindre effort. Dans le monde libre, nous trouvons un certain nombre de logiciels qui contribuent significativement à l'atteinte de notre objectif sans l'atteindre totalement. Profitant d'être en mesure d'accéder au code et le modifier, nous avons principalement sélectionné deux frameworks : le premier, Django (qui lui même repose sur de nombreux logiciels libres) spécialisé dans la gestion des modèles et des applications, le deuxième, Sencha-ExtJs, spécialisé dans les interfaces utilisateurs et une gestion dynamique des composants Web. Sur la base de ces deux piliers, nous avons construit l'infrastructure pour la définition et l'exécution des prototypes.

2.6.1 Modélisation Agile, mais d'abord prototypage

Un modèle est une abstraction[21] qui décrit une ou plusieurs aspects d'un problème ou une solution potentielle face à un problème. Traditionnellement, les modèles sont considérés comme d'un ou plusieurs diagrammes ainsi que de toute la documentation correspondante. Cependant, les artefacts non-visuels tels que des collections de cartes CRC [20], une description textuelle d'une ou plusieurs règles de gestion, ou la description structurée d'un processus d'affaires sont également considérés comme des modèles.

Une méthode agile[93] [40]est avant tout itérative sur la base d'un affinement progressif des besoins satisfaits par des fonctionnalités en cours de réalisation et même déjà réalisées.

Le pratiques de développement Agile rendent la communication plus efficace [51], et favorisent ainsi le processus de création de connaissances. Contrairement à la croyance populaire, l'architecture (dont la modélisation) est un aspect important des efforts de développement de logiciels agiles, tout comme les efforts traditionnels, est une partie essentielle pour répondre aux besoins du monde réel.

Un modèle agile[21] (Agile Model - AM), est un modèle qui répond à son objectif et rien de plus ; il est compréhensible aux parties prenantes, est simple, suffisamment précis, cohérent et détaillé, et l'investissement dans la création et l'entretien apportent une valeur positive au

20. CRC : Class Responsibility Collaboration

projet. La modélisation Agile [73] (AM) n'est pas un processus normatif. En d'autres termes, elle ne définit pas les procédures détaillées sur la façon de créer un type de modèle, à la place, elle fournit des conseils pour savoir comment être efficace en tant que modélisateur.

Certaines personnes croient que le développement agile signifie "pas de conception", alors qu'en fait, la modélisation agile, favorise une rétroaction rapide. Un rétroaction rapide signifie obtenir quelque chose en face de l'utilisateur le plus rapidement que possible. Cela signifie que les prototypes qui peuvent être rapidement mis sur pied ont une grande valeur en termes de recueil précoce de la rétroaction des utilisateurs.

Nous considérons que pour être réellement agile, le modelés doivent permettre d'anticiper le fonctionnement du système ciblé. Un tel modèle est un prototype. Depuis longtemps, le prototypage a été une technique de collecte d'informations utile pour compléter le cycle de vie du développement des systèmes traditionnels. Les méthodes Agile préconisent la communication directe avec le client. Donc, les analystes des systèmes doivent travailler systématiquement à obtenir et évaluer les réactions des utilisateurs et a recevoir les suggestions pour les ajouts et / ou suppressions de caractéristiques. Analogue à la philosophie de « refactoring »[72][75], la démarche proposée est de prendre une approche itérative et incrémentale à la réalisation de prototypes.

Kendall [93] considéré qu'il y a quatre conceptions de prototypes :

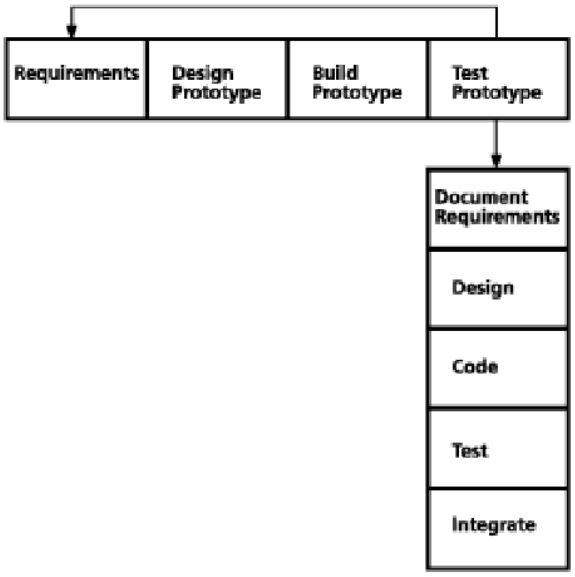
- A. prototypes rafistolées (Patched-up)
- B. maquettes non opérationnels
- C. premiers modèles à grande échelle
- D. prototypes qui ne contiennent que quelques-unes des caractéristiques essentielles du système

Le prototype c'est un artefact que l'on fait parce que l'on a besoin de réponses, et de partager notre compréhension des besoins. Pour qu'un prototype soit valable dans notre contexte, on considère qu'il devrait répondre a la catégorie « D » de kendall. Soit des prototypes qui ne contiennent que quelques-unes des caractéristiques essentielles du système.

L'utilisation des prototypes automatiques à partir de modelés est considérée comme le prochain niveau d'abstraction, dont il y a beaucoup d'investissement[77] [29]. Pourtant, la création des archétypes [21] [113], la transformation de modèles et la génération de codes deviennent des processus qui ne sont pas facile à maîtriser, et en date d'aujourd'hui (2013) les spécifications de OMG sont encore en version BETA[125] [22].

21. Le terme archétype est utilisé pour indiquer les modèles de niveau de connaissances qui définissent des structures d'information valides. Les archétypes définissent les règles pour traduire les modèles dans une application particulière [30].

22. La version 1.1 viens juste de sortir au moment de la révision de cet document. On n'a eu pas le temps de lire le document.[126]



Several successive prototypes

FIGURE 2.10: The Prototyping life cycle model (Dorfman 2000) [62])

On propose une démarche de modélisation Agile basée sur la méthode Datarun et la construction de prototypes interactives. Ce processus peut aider à réduire le risque global et permet au projet de s'adapter rapidement aux changements.

La clé de cette technique est que les concepteurs ont un contrôle complet sur la forme et la fonction du prototype, et peuvent obtenir une rétroaction utile sans aucun code en cours d'écriture. En ce qui correspond à la présentation du prototype, il devrait y avoir une interface simple tout aussi efficace pour les tests d'interaction.

2.7 Conclusion

En conclusion, nous retenons le cycle proposé par Dorfman et Thayer (2000)[62] qui invoque la phase de prototypage comme un cycle à l'intérieur (début) du cycle de développement. Si le prototypage est fait avec soin, le résultat final est le cahier des charges précis du système final.

Parallèlement, Pascot a résume la méthode Datarun [128][132] comme « une démarche simple et rapide pour la réalisation des dossiers de conception fonctionnelle avec un prototype ». La méthode Datarun, est une démarche structurée qui permet au concepteur d'étudier les opérations, et les décisions importantes dans l'organisation, afin de trouver les données primaires essentielles au fonctionnement de l'organisation (les intrants du SI) pour ensuite construire progressivement un système informatique qui satisfasse les besoins des décideurs. La phase de prototypage permet de tester de façon très réaliste, avec le concours actif des utilisateurs,

l'utilisation et l'utilisabilité d'un produit. Dans notre exposition on met en évidence, comment l'idée centrale de la composition d'objets est progressivement remplacée par la notion de transformation de modèles[65]. On peut voir cela en continuité ou en rupture. Pourtant, l'idée de que les systèmes logiciels sont composées d'objets interconnectés n'est pas en opposition avec l'idée du cycle de vie du logiciel étant considéré comme une chaîne de transformations de modèles.

La démarche suivie dans Datarun[129] isole la spécification de la construction. D'abord elle permet de définir ce que doit faire le système globalement, puis en détail, avant de le réaliser. Datarun définit la transformation des modèles, à partir d'un modèle conceptuel (MCD), qui est transformé dans un modèle structural (MSI) lequel servira de base pour la définition des composants de haut niveau et qui aura la responsabilité de l'exécution du système. Nous combinons une vision d'analyse avec une approche technique moderne basée sur l'état de l'art de la technologie. Le résultat est un cycle de développement enrichi, basé sur la collaboration client-concepteur pour produire un langage précis basé sur l'exécution de modèles. Le langage en question est la création de prototypes à partir des besoins.

La plupart des auteurs veulent que leurs méthodologies puissent être universelles **[88] [73]**, sans comprendre nécessairement les conditions et les limites de leur façon de faire. Chaque méthode a ses points forts et ses limites, en effet elles sont appropriées pour des types de projets spécifiques. La méthode que nous proposons s'adresse à l'analyse de systèmes très orientée vers le traitement de données, avec beaucoup d'interaction avec l'utilisateur.

Chapitre 3

Spécification déclarative

« Bien souvent, le défi avec les modèles n'est pas de faire en sorte qu'ils soient suffisamment complets, mais de les rendre les plus simples et les plus compréhensibles possible. »
(Avram et al -2006)[29]

3.1 Introduction

La spécification déclarative est une méthode de conception qui met l'accent sur le «quoi» (règle) et « que » faire (le résultat), elle s'oppose à la déclaration procédurale ou impérative qui porte sur le «comment» le faire (le processus). Par exemple, les pages HTML sont « déclaratives » car elles décrivent ce que contient une page (texte, titres, paragraphes, etc.) et non comment les afficher (positionnement, couleurs, polices de caractères, etc.). Alors que dans l'approche « impérative » on décrit le comment, c'est-à-dire la structure de contrôle des étapes nécessaires pour atteindre la solution. Tout système repose sur une architecture d'information, c'est avant tout un ensembles de représentations du réel perçu (souvent appelé contenu sémantique), on peut l'exprimer par un modèle conceptuel de données, s'il est normalisé il assure une la qualité de la représentation avec la mise en œuvre des règles d'intégrité référentielle. Un modèle normalisé de l'ensemble des données et une description de chacune des interfaces (accès aux données en saisie ou consultation) sont un excellent point de départ pour spécifier « quoi » faire. Dans ce chapitre, on regarde différentes approches de spécification basées sur des modèles et on définit ce qui sera la base de notre développement.

Nous étudions dans ce chapitre différentes tendances et travaux tant académiques que pratiques, nous avons essentiellement retenu les travaux de OMG[125], de Eric Evans[65], de Bézivin[38], de Den Hann [54] et de Smolìk [150]. Puis, en nous appuyant sur les idées retenues

dans Datarun par Pascot[129] nous proposons une vision de conception consistant en une spécification déclarative implémentée par un ensemble de règles et un cadre d'exécution.

3.2 Conception guide par modèles

« Smart data structures and dumb code works
a lot better than the other way around »
(Raymond 2005) [137]

La conception guidée par le modèle est une approche de développement logiciel qui vise à développer des logiciels à partir de modèles en utilisant les principes de développement AGILE. Il y a plusieurs variations de cette approche, entre autres on peut mentionner : Model-driven engineering MDE, Model-Driven Architecture MDA, Domain Driven Design DDD, Model-Driven Software Development, Domain analysis, Meta modeling, Model-driven generation, etc.

Cette approche est basée sur l'idée que les modèles sont les artefacts primaires (fondateurs) à partir desquels d'autres artefacts sont générés.

- Le modèle doit être lisible et interprétable par l'ordinateur. Le modèle n'est pas un simple graphique.
- La lisibilité et l'interprétation des modèles est une condition préalable pour être en mesure de générer des artefacts[152].
- Pour interpréter les modèles, il est nécessaire définir un langage, ce langage est aussi un modèle qui est appelé métamodèle.

L'idée centrale de la composition de l'objet est progressivement remplacée par la notion de transformation de modèles. On peut voir ces propositions en continuité et non en rupture. L'idée de que les systèmes logiciels sont composés d'objets interconnectés n'est pas en opposition avec l'idée du cycle de vie du logiciel étant considéré comme une chaîne de transformations de modèles[65].

Les créations de composantes et de cadres (frameworks) spécialisés permettent de pré-fabriquer de plus en plus de composantes techniques complexes (briques), de sorte que la conception peut maintenant se concentrer sur le domaine et les problèmes métier. Ici, l'espoir est d'améliorer considérablement la productivité de la production et la qualité des systèmes d'information.

La conception de logiciels est un art[42], et comme tout art, il a besoin de se matérialiser. Tout comme les peintres qui ont besoins des bonnes peintures et bonnes toiles, la matérialisation d'un système sous forme de code, grâce à l'évolution des outils informatiques permet de focaliser l'attention sur la création et non sur les composantes logicielles et matérielles.

L'importance pour la conception guidée par des modèles est de focaliser sur le domaine et non sur les aspects techniques. De la même façon que l'artiste connaît son matériel, le concepteur doit connaître intimement ses outils évitant ainsi que les concepteurs passent trop de temps avec les aspects techniques. Il y a particulièrement deux travaux qui on retenu notre attention « MDA » et « DDD » voici un résume des idées les plus pertinents pour notre projet :

3.2.1 MDA

En novembre 2000, l'OMG a rendu public l'initiative MDA[**?**][123]. L'objectif de cette initiative est fondé sur les avantages qui pourraient être tirés d'un passage de la réalisation ad-hoc du code à des pratiques fondées sur des modèles qui d'une manière directe ou indirecte utilisent des composants et des patrons (patterns) de comportement. L'idée promue par MDA[32] consiste à utiliser des modèles pour le développement de systèmes, augmentant ainsi le niveau d'abstraction dans la spécification des programmes. Les modèles de niveau supérieur sont transformés en modèles de niveau inférieur jusqu'à ce que le modèle puisse être rendu exécutable en utilisant soit la génération de code soit l'interprétation du modèle[37]. Le passage d'un niveau à l'autre implique l'ajout (si possible automatique) des éléments propres au nouveau niveau, mais ignorés au précédent [1].

Dans cet enchaînement des niveaux, le principe « *Tout est objet*» a été très utile en apportant simplicité, généralité et puissance. De même, en MDA, le principe fondamental selon lequel « *Tout est un modèle* » procure de nombreuses propriétés intéressantes.

L'idée d'un modèle exécutable permet de concevoir le SIO directement en fonction des concepts du domaine cible. Cela permet aux experts du domaine de comprendre sans interprétation ou déformation les fonctionnalités requises du point de vue des usagers. Ainsi, des experts du domaine peuvent être en mesure non seulement de valider la satisfaction de leurs besoins, mais ils peuvent aussi apprendre à modéliser le système par eux-mêmes. Suivant cette approche, idéalement un système d'information informatisé devrait pouvoir être modélisé directement par les experts du domaine qui en ont besoin.

Dans cette approche, l'étape suivant la modélisation est la construction d'un ensemble de règles (conception déclarative) dans lequel une description précise des propriétés est suffisante pour contrôler les aspects les plus importants d'un SIO. Autrement dit, on peut passer d'un modèle à un meta-programme pour spécifié le SIO.

Pourtant, il y a ses écueils dans la pratique [37], par exemple :

- Les langages de déclaration ne sont pas suffisamment complets pour spécifier tout ce qu'il faut dans un SIO (et les rendre plus complets les éloigne de l'objectif initial).

1. Rappelons ici qu'abstraction dans un niveau signifie ignore des aspects qui sont pris en compte dans un autre niveau

– Les frameworks rendent normalement très difficile d'étendre le logiciel au-delà de la partie automatisée.

– Les détails d'implantation qui sont ajoutés après la génération du code sont faciles à perdre s'ils ne sont pas remontés au modèle.

MDA est basé sur des standards chapeautés par l'OMG tels que : UML, MOF, XMI, etc. leur objectif principal est de traduire des modèles en code exécutable. Le MDA a était introduit en 2001, et aujourd'hui (2013) l'industrie de logiciels ne l'a pas adopté massivement [148]. Cependant, nous avons une forte proximité avec MDA dans les orientations : « Métadonnées, Services communs et Orientation axées sur le savoir (ontologie) »

3.2.2 Domain Driven Model (DDD)

La conception pilotée par le domaine (DDD)[65] est une approche de conception de logiciel basée sur ces principes [2] :

1. Placer objectif principal du projet sur le domaine central et la logique de domaine.
2. Le domaine s'exprime au moyen de modèles ;
3. Une collaboration « AGILE » entre les experts métier (domaine) et les concepteurs pour affiner itérativement le modèle de domaine.

Le terme DDD a été forgé par Eric Evans (2004) dans son livre « Domain-driven Design : Tackling Complexity in the Heart of Software ». Selon Evans le monde autour de nous est beaucoup trop complexe pour nos capacités de manipulation [3]. Nous avons besoin d'organiser l'information, de la diviser en petits morceaux, de regrouper ces morceaux en modules logiques, et en prendre un à la fois pour après revenir à la totalité pour garantir la consistance générale [4].

DDD identifie un ensemble de modèles de conception et un style d'architecture spécifique. DDD est aussi, par conséquent, une approche pratique pour la conception de logiciels basée sur l'importance du domaine d'activité, ses éléments, les comportements et les relations entre eux. Voici les concepts exprimer pour DDD pertinents pour notre travail :

Langage omniprésent

Un des concepts de base de l'approche DDD est l'identification d'un «Langage omniprésent (Ubiquitous Language) » [5]. Le « Langage omniprésent » doit former un langage commun apporté par des experts du domaine pour décrire les exigences du système, et qui fonctionne

2. Cette section est basée entièrement dans les travaux de Evans[65] .
3. En coïncidence avec LeMoigne[103]
4. Les arguments de base de la conception analytique et la conception systémique[47]
5. Le langage omniprésent est différent de l'idée de DSL « Domain spécification language » qui est plutôt une représentation technique. Par exemple HTML est considéré un DSL[155][74].

aussi bien pour les utilisateurs, pour les commanditaires que pour les développeurs[6]. Souvent, les problèmes de communication des exigences fonctionnelles proviennent des malentendus et d'un langage ambigu.

Modèle

Selon Eric Evans, le modèle est une représentation interne du domaine ciblé, et il est nécessaire tout au long de la conception et du processus de développement. Le modèle est l'essence même du logiciel, on doit créer des façons de l'exprimer pour communiquer avec les autres. Le modèle doit être communiqué, et Evans soutient qu'il y a deux formes de partage de la connaissance d'un modèle, la première est graphique (diagrammes, cas d'utilisation, dessins, photos, etc.), l'autre est textuelle. Nous y ajoutons une troisième : des prototypes fonctionnels.

Diagrammes

> « A lot of object model diagrams are too complete
> and, simultaneously, leave too much out. »
> (Evans 2004) [65, P31]

Un autre caractéristique de DDD est que (comme dans les pratiques « AGILE ») on y met l'accent sur les diagrammes moins formels et les conversations casuelles. Evans soutient que UML est un excellent outil. Mais, il n'est pas suffisent pour tous les cas, et il faut retourner aux dessins et aux textes pour assurer une bonne communication.

Architecture 4 couches

La plupart de code dans un système n'a rien a voir directement avec le domaine, il y a des aspects comme les détails de l'interface d'utilisateur, les services transversaux comme la sécurité, la gestion de la persistance, etc. Il est fréquent que plusieurs de ces aspects soient intégrés dans les objets métier, et que la logique métier soit distribuée dans les composantes d'interface ou dans la gestion de la persistance. En conséquence, lorsque le code du domaine est mélangé avec les autres couches, des changements superficiels à l'interface utilisateur peuvent réellement changer la logique métier. Nous considérons que pour la « conception » en ce qui concerne le comportement du système tel que le perçoit l'usager en regard de son contenu doit se faire dans une couche conceptuelle (le domaine). La division en couches dans le logiciel permet ensuite l'implémentation de la solution. Pour la création de l'outil de prototypage nous retenons l'approche de conceptualisation en couches.

6. Attention, la création d'un tel langage commun n'est pas simplement un exercice d'acceptation des termes auprès d'experts du domaine et leurs applications.

Modules

Les modules sont une ancienne pratique dans le design de système d'information. Si le modèle raconte une histoire, les modules en sont les chapitres. Pour une application assez large, le modèle atteint un point où il est difficile à considérer dans son ensemble, et comprendre les relations et les interactions entre les différentes parties devient difficile. Pour cette raison, il est nécessaire d'organiser le modèle en modules.

Lors du choix de modules, il faut se concentrer sur la cohésion conceptuelle à l'intérieur de module et chercher un couplage faible dans le sens des concepts qui peuvent être compris et raisonnés indépendamment les uns des autres. Si les modules sont bien utilisés dans les projets, il est plus facile d'obtenir l'image d'un grand modèle. C'est un moyen simple et efficace pour gérer la complexité d'un grand système.

Il faut distinguer les concepts de cohésion communicationnelle et cohésion fonctionnelle. La cohésion communicationnelle est atteinte lorsque les pièces du module fonctionnent sur les mêmes données, il est logique (techniquement) de les regrouper, car il y a une forte relation entre eux. La cohésion fonctionnelle est obtenue quand toutes les parties du module travaillent ensemble pour accomplir une tâche bien définie. Ceci est considéré comme le meilleur type de cohésion.

Les modules doivent avoir des interfaces bien définies pour être accessibles par d'autres modules. Cette interface doit être présentée en forme de services.

Pour nommer les Modules. Il est important de refléter la vision du domaine. Autrement dit, le nom de module fait partie du « langage omniprésent » mais en même temps le choix du type de module dépend de la réalisation.

Conclusion DDD

En principe DDD est orienté vers des domaines non triviaux, les principes de base sont simples, mais l'implémentation n'est pas triviale. Même, si Evans affirme que DDD n'est pas lie à l'analyse objet, il y a des aspects de modélisation qui ont une forte relation avec l'implémentation objet (fabriques, répertoires, etc) ce qui peut nuire la compréhension des experts du domaine. Nous adoptons l'idée de base de DDD, tout en intégrant la méthode traditionnelle Datarun qui est une démarche plus orientée vers la simplicité sans perdre de vue la complétude de la solution.

3.3 Méta-modélisation

«Metamodeling means identification of general concepts
that exist in a given problem domain and their relations. »
(Smolik 2006)[150]

Un modèle est une partie essentielle de la conception de logiciels. Nous en avons besoin pour être en mesure de composer avec la complexité[99]. Notre processus de réflexion sur le domaine est synthétisé dans un modèle, mais nous ne sommes pas seuls dans ce processus, nous avons donc besoin de partager des connaissances et des informations, et nous devons le faire bien, précisément, complètement et sans ambiguïté. Un modèle est une façon d'exprimer et de communiquer avec nous même, mais surtout avec les autres[99].

«Meta» signifie littéralement «après» en grec. En informatique, le terme est largement utilisé avec plusieurs significations différentes[119] : pour les données, métadonnée signifie donnée sur les données et réfère aux dictionnaires de données, référentiels, etc. Dans les langages de programmation, metainterpreter est un interpréteur d'un (programme) interprète ; dans la modélisation conceptuelle, méta-modèle est un modèle d'un modèle de données, il en reflète les composantes, la structure et les règles.

Les données sont modélisées par des métadonnées (entité, attribut, ...) qui composent un métamodèle ; ces unités peuvent être des instances de méta-données qui sont au leur tour partie d'un méta-métamodèle, et ainsi de suite. On peut avoir des métamodèles qui sont autodescriptifs à un niveau arbitraire d'autodescription[119].

MetaDonnées	Données (une instance de ..)
Étudiant	Étudiant
Nro Id	909 302 123
Nom	Gomez
PreNom	Dario

En résume, la méta-modélisation consiste [150] à l'identification de concepts généraux et leurs relations qui existent dans un domaine de problème spécifique. Le défi est de trouver un langage suffisamment simple et complet qui puisse être utilisé pour définir les exigences du domaine. Faire et opérationaliser un méta-modèle est une partie importante de notre projet.

3.3.1 Point de vue sur les méta-modèles

La norme ANSI X3.138-1988[127] propose une standardisation indépendante de la technologie pour la définition d'un dictionnaire de ressources de l'information « IRDS » (information resource dictionary system) similaire au modèle ER. divisé en 4 niveaux différents de données (figure 3.1) :

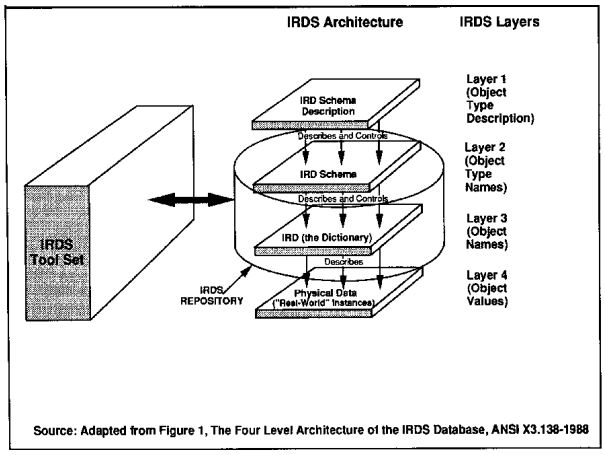


FIGURE 3.1: IRDS Architecture selon Ansi X3.138-1988[127]

- Niveau 1 - différents types de schémas IRDS, par exemple, des schémas de langage de programmation, modèles.
- Niveau 2 - schéma pour le dictionnaire de données, par exemple, ce qui est une procédure , ce qui est une variable, etc.
- Niveau 3 - dictionnaire de données pour les données d'application, par exemple, les identifiants des procédures, variables, types de données, etc.
- Niveau 4 - données d'application, par exemple de code de logiciels ;

Bien que la méta-modélisation soit une base essentielle pour le développement guidé par le modèle, l'interprétation traditionnelle comme un "langage de définition" de la métamodélisation ne répond pas à toutes les exigences techniques pour une infrastructure MDD (Model Driven Development)[27]. Toutefois, il peut facilement être étendu pour fournir le soutien nécessaire pour MDD .

La motivation sous-jacente de MDD est d'améliorer la productivité de deux façons [27] :

- améliore la productivité à court terme de développeurs en augmentant la valeur d'un artefact logiciel en termes de combien de fonctionnalités qu'il offre.
- améliore la productivité à long terme de développeurs en réduisant la vitesse à laquelle un artefact logiciel devient obsolète.

La figure 3.2 illustre l'infrastructure traditionnelle à quatre couches qui sous-tend la première génération de technologies MDD correspondantes à « Unified Modeling Language »[124] et « MetaObjectFacility »[122].

Cette infrastructure se compose d'une hiérarchie de niveaux de modèles, chacun (sauf le haut) étant caractérisé comme une instance de niveau supérieur.

- Le niveau inférieur, M0, contient les données de l'utilisateur. Des objets de données réelles que le logiciel doit manipuler.

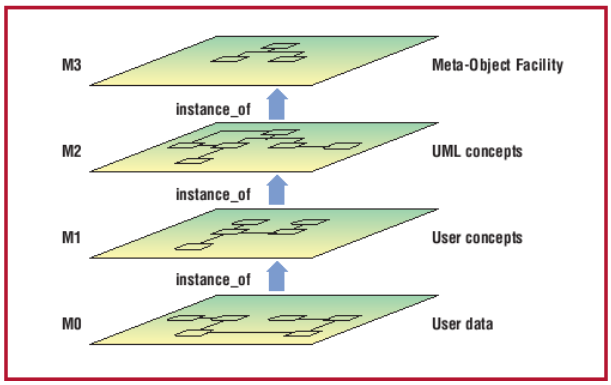


FIGURE 3.2: Traditional Object Management Group modeling infrastructure[27]

- Le niveau suivant, M1, est conçu pour maintenir un modèle des données de l'utilisateur (M0).
- Le niveau M2 détient un modèle de l'information de M1. Parce que c'est un modèle d'un modèle, il est souvent considéré comme un métamodèle.
- Le niveau M3 détient un modèle de l'information au M2, et est donc souvent appelé le méta-métamodèle. Pour des raisons historiques, il est également considéré comme le Meta-Object Facility (MOF)[122][27].

Cette architecture à quatre couches a l'avantage de facilement accueillir de nouveaux standards de modélisation par exemple, le métamodèle « Common Warehouse » pour les entrepôts de données.

Les outils MOF peuvent donc soutenir la manipulation des nouvelles normes de modélisation et de permettre l'échange d'informations à travers les standards de modélisation compatibles MOF. Bien qu'elle ait été une fondation réussie des technologies MDD de première génération, cette infrastructure traditionnelle ne s'adapte bien pour gérer toutes les exigences nécessaires pour améliore le rapport obsolescence des artefacts logiciels[27].

Une grande partie des travaux sur l'amélioration de l'infrastructure de métamodélisation a mis l'accent sur l'utilisation de méta-modélisation comme un outil de définition de langage. Avec cet accent, l'instance de relation linguistique est considérée comme dominant et le niveau M2 et M3 sont considérés comme des couches de définition de langage. Cette approche relègue instance ontologique de relations (types de domaine), à un rôle secondaire.

La figure 3.3 montre cette dernière interprétation de l'architecture à quatre couches selon les nouvelles normes MOF 2.0 et UML 2.0[27]. Bien qu'il garde encore une vision essentiellement linguistique, il commence à présenter une préoccupation ontologique. Les différentes nuances de niveau M1 dans la figure suivante indiquent l'existence de deux « meta-dimensions » orthogonaux explicites. Elle illustre la relation entre les éléments du modèle et les éléments du monde réel correspondant.

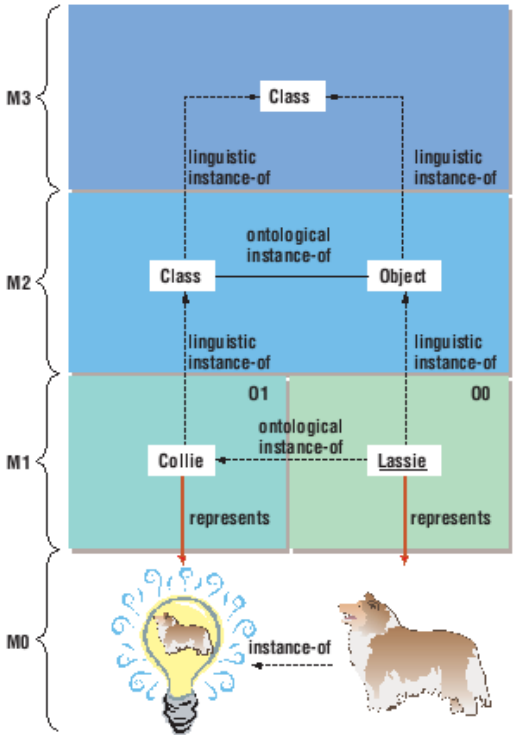


FIGURE 3.3: Linguistic metamodeling view.[27]

Plusieurs auteurs [27] [17] [108] [150] ont différentes approches pour aborder la problématique de méta-modélisation. On a choisi l'approche de Smolik (2006) à cause de sa vision claire et indépendante d'autres outils tels que l'UML ce qui le rendre plus léger et facile à utiliser (figure 3.4).

3.4 Le Prototypage comme une Meta-Solution

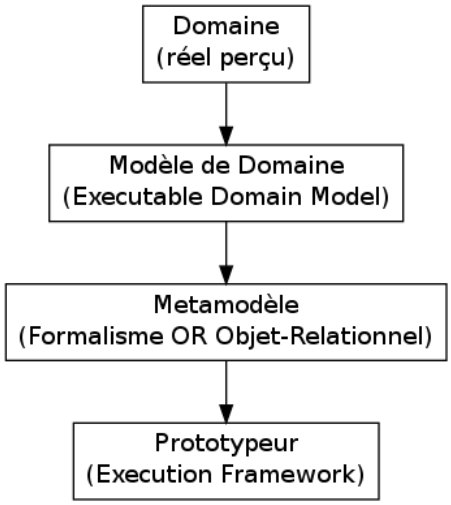


FIGURE 3.4: Implémentation de métamodélisation retenu

Le "**modèle de domaine**[7]" correspond à la définition conceptuelle des exigences d'un domaine spécifique et permet aux experts du domaine d'être en mesure de parler directement en termes du domaine (réel perçu). Le modèle de domaine doit décrire parfaitement les exigences fonctionnelles du système dans un domaine spécifique et suivant notre objectif il doit être directement exécuté comme un logiciel qui reflète ces exigences. Dans notre projet le modèle de domaine exécutable correspond au formalisme "objet-relationnel OR" qui serait expliqué plus tard (5.2).

En complément, il doit exister un cadre d'exécution qui doit abstraire à la fois les détails de la technologie de mise en œuvre et les règles générales inhérentes au domaine [65] [150] . Le cadre d'exécution fournit un environnement dans lequel le modèle de domaine exécutable peut vivre et qui prévoit la mise en œuvre concrète de l'ensemble des règles générales du domaine qui ne font pas partie du modèle. Notre cadre d'exécution correspond au prototypeur.

Finalement on a la définition de notre méta-modèle qui est notre langage de modélisation spécifique pour la définition des prototypes des systèmes informatiques orientée données.

La portée de ce qui peut être défini dans un modèle est spécifiée par le métamodèle. Ainsi on a défini les notions de "Projet", "Modèle", "Concept", "Propriété" et "Relations" sur lesquelles on reviendra plus tard (5.2). Notre méta-modèle spécifie quels éléments peuvent être contenus dans le modèle et comment ils se rapportent entre eux. En réalité,le méta-modèle est une spécification d'un langage de modélisation particulier au domaine utilisé pour exprimer les exigences du système et définir exactement comment il doit les réaliser.

7. On adapte les définitions des termes "Executable Domain Model" , "Execution Framework" et "Metamodel" realisés par Smolik (2006)[150]

Le méta-modèle doit offrir assez de puissance pour décrire la plupart des exigences du domaine alors qu'il a besoin de faire abstraction de tout ce qui n'est pas lié directement au domaine ou tout ce qui est une règle générale dans le domaine. En autres mots, notre méta-modèle doit être capable de rendre explicite tout ce qui est implicite dans le domaine.

La création d'une « Méta-Solution » par moyen de la méta-modélisation se concentre principalement sur l'objectif de logiciel lui-même et permet de se concentrer sur l'essence de système à mettre en œuvre.

La construction de notre méta-modèle doit distinguer ce qui est variable (donnés) et ce qui est une règle (connaissances) ainsi le prototype devient une vision informatique sur le réel perçu (information).

Les aspects variables doivent être capturées dans les modèles et les règles générales sont mis en œuvre par le prototypeur en utilisant le méta-modèle.

En résumé et conclusion voici la définition des termes plus importants de notre démarche de méta-modélisation (figure 3.4) :

- **Modèle de domaine (Executable Domain Model**) : Le modèle de domaine est un modèle conceptuel qui décrit totalement les exigences fonctionnelles (du système cible) dans un domaine spécifique. Il doit être directement exécuté comme un SIO. Ce modèle doit être aussi simple et minimal que possible tout en exprimant suffisamment tout le nécessaire. Le modèle exécutable de domaine doit abstraire complètement les détails de la technologie de mise en œuvre. Le modèle de domaine doit être fait à l'aide de formalisme « modèle OR » développé dans la section 5.2
- **Méta-modèle** : Le méta-modèle correspond à la définition des éléments de conception d'un système d'information orientée donnée. Le méta-modèle correspond au formalisme « modèle OR » et permet d'être en mesure de parler directement en termes de situations de domaines spécifiques.
- **Méta-donnée (Metadata) :** Les métadonnées proviennent de modèles conceptuels de données produites avec les règles établies par le méta-modèle.
- **Prototypeur (Execution Framework)** : Le prototypeur fournit un environnement dans lequel le modèle de domaine peut vivre. Il doit diminuer le nombre de paramètres qui doivent être définis d'une manière générique par défaut (métadonnées). Il doit se concentrer sur la définition des aspects importants de domaine en isolant les détails spécifiques de la technologie de mise en œuvre.

L'objectif principal c'est de séparer l'exécution dynamique (prototypage) de la spécification statique des contenus (modèle de domaine) par moyen d'un méta-modèle. Dans notre environnement de prototypage, le modèle devient simplement l'abstraction de la représentation exécutable de SIO .

Chapitre 4

Évolution du projet

Dans les chapitres 2 et 3, nous développons le cadre théorique de notre recherche. Dans le chapitre 2, nous posons la problématique de la recherche en ayant recours à la littérature dans le domaine de l'informatique. Nous avons en particulier souligné que la résolution de la problématique de conception passe par l'utilisation de modèles et de prototypes. Dans les chapitres suivants nous allons faire un parcours l'évolution du projet. Le développement de l'outil et un résumé des caractéristiques de l'outil final incluant un guide d'utilisation .

L'objectif final est de produire un outil informatique qui soit capable de recevoir un modèle conceptuel de données [1] pour produire un prototype fonctionnel. Donc, cette partie du projet est plutôt pratique. Pourtant, on va essayer de faire dans ce document une abstraction des aspects technologiques pour expliciter les concepts de base, habituellement implicites et flous, qui sont en arrière de la discipline informatique.

Notre objectif est d'utiliser des outils open source pour diminuer l'intervalle entre le modèle et la façon dont les utilisateurs perçoivent la solution à ses besoins. Dans ce projet de recherche, l'utilisation de métadonnées sera étendue afin de supporter les fonctions d'applications basées sur des vues du modèle conceptuel de données. La première étape de la recherche est de construire un métamodèle avec une définition ouverte (dans un langage de balisage standard comme XML ou JSON) afin de permettre la communication avec d'autres outils.

- Nous avons commencé par créer un Plugin pour l'outil Open ModelSphère[9] afin d'exporter les modèles conceptuels de donnée.
- La deuxième étape fut d'implémenter le support pour le modèle de spécification d'interface (MSI Datarun) dans le métamodèle.
- La troisième étape est de transformer la représentation interne du méta-modèle en un prototypeur basé sur les vues de modèle. Cette étape sera la contribution principale

1. Les futures références au « modèle conceptuel » ou simplement « modèle de domaine » donne pour acquis qu'il respecte les normes du métamodèle défini dans le cadre de ce travail

du projet de recherche. Il était prévu d'interfacer avec l'outil Modelibra[138] un projet OpenSource de Dzenan Ridjanovic.

Dans la planification initiale, nous n'envisagions pas de faire le développement et la spécification du modèle d'interface dans l'outil OMS (Open ModelSphere) mais de reprendre l'outil Modelibra ou de construire une application web pour produire le prototype. L'option de l'application Web était pas toujours envisageable, car elle semblait très compliquée et coûteuse. Pourtant, on était clair dans l'objectif : la production d'un outil de prototypage.

4.1 Unicité de paradigme

Les mathématiques peuvent être modélisées et facilement mises en œuvre en utilisant la programmation procédurale. Parce que beaucoup de théories mathématiques sont simplement traitées à l'aide d'appels de fonctions et de structures de données. Les langages procéduraux offrent un support limité pour la conception dirigée par modèles. Ces langages ne proposent pas les structures nécessaires à l'application des éléments clés d'un modèle. Un tel langage ne peut pas encapsuler facilement les liens conceptuels entre le domaine et le code. La mise en œuvre pour attacher solidement le code aux modèles généralement exige des outils et langages de développement de logiciels pour soutenir le paradigme de modélisation.

L'approche donnée permet de découvrir l'information circulant dans l'organisation et la représente par moyen d'entités et de relations. L'algèbre relationnel a été inventée en 1970 par Edgar Frank Codd [50] [49]. Cet algèbre est constitué d'un ensemble d'opérations formelles sur les relations (qui sont en fait les tables ou entités). L'idée de base est que les opérations relationnelles permettent de créer une nouvelle relation (table) à partir d'opérations élémentaires sur d'autres tables (par exemple l'union, l'intersection, ou encore la différence). L'algèbre relationnelle est une théorie mathématique dérivée de la théorie des ensembles qui définit des opérations qui peuvent être effectuées sur des relations. Les relations sont des matrices contenant un ensemble de n-uplets.

L'approche orientée objet est basée sur des principes d'ingénierie de logiciels éprouvés. L'approche relationnelle est basée sur des principes mathématiques aussi solides. Puisque les paradigmes sous-jacents sont différents, les deux technologies ne fonctionnent pas ensemble de façon transparente [2].

La conception orientée modèle est basée sur la vision objet. Pourtant, les modèles ne doivent pas nécessairement être des modèles objet, ni la conception par modèle est non plus spécifique à la programmation orientée objet. Mais, on peut faire un meilleur travail lorsque la technologie de mise en œuvre prend directement en charge le paradigme de modélisation [28]. En définitive nous pensons qu'il est toujours préférable de maintenir un paradigme unique, donc on utilise

2. Au début des années 1990, les différences entre les deux approches a été appelé « désadaptation d'impédance (impedance mismatch)[23]

l'approche objet et nous nous servons des outils qui permettent l'adaptation transparente vers le modèle relationnel [3].

4.2 Historique de développement

4.2.1 Open ModelSphere (OMS)

Il y avait deux volets de travail sur OMS, le premier était l'extraction de modèles et le deuxième était la possibilité de le faire évoluer pour compléter la définition du modèle d'interface.

La première partie, le développement de l'outil d'extraction a été fait avec la collaboration de Marco Savard et de l'entreprise NeoSapiens, ils nous ont fourni la structure de classes en java ainsi que la façon d'installer des plugins sur OpenModelsphere. Les détails du développement de l'outil d'extraction ne sont pas pertinents pour l'objectif de construction d'un prototypeur. Ce qu'on retient de cette expérience :

- La création d'OMS date de 2002, il était développé à partir de zéro, il n'utilise pas de frameworks standards.
- Le dessin d'OMS était très ambitieux pour son époque. En conséquence la structure de classes est vaste et la courbe d'apprentissage pour le développement est très lente.

En conséquence, nous avons fait le plugin d'extraction (voici un exemple 4.1) et nous abandonnons l'idée de continuer le développement sur ModelSphere.

4.2.2 Modelibra.

Modelibra[138] est une famille de logiciels open source qui est utilisé pour développer des applications web dynamiques basés sur des modèles de domaine. Ce projet est constitué d'un outil de conception graphique, un cadre « modèle » de domaine, un boite à outils de composants Web, une collection de déclarations CSS et XML. À partir d'un module de dessin graphique, l'outil fait la création d'une base de données et la génération de code Java pour l'administrer. L'objectif de cette étape était d'utiliser Modelibra et se concentrer sur la réalisation des vues de modèle essentielles. Les objectifs fixés étaient à ce moment :

1. Comparer les méta modèles de Modelibra et Open ModelSphere et décider s'il fallait augmenter les méta modèles de Modelibra avec les caractéristiques du méta modèles de Open ModelSphere.
2. Ajouter le support pour les vues de modèle au méta modèles de Modelibra.
3. Transformer la représentation interne du méta modèle de Open ModelSphere en fonction du méta-modèle Modelibra.

3. Le **ORM** acronyme « d'object-relational mapping » est une technique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle.

Algorithme 4.1 Exemple de document d'extraction de OMS Version 01

```
<?xml version="1.0" encoding="utf-8"?>
<project name="Exemple de raccordement">
  <udfs>
    <udf alias="precision" description="" name="PRÉCISION" type="Multiline Text" />
  </udfs>
  <datamodel idmodel="1" idref="0" name="Modèle Conceptuel Corporatif">
    <tables>
      <table alias="" name="CONTACT" physicalName="" superTable="">
        <column alias="" foreign="False" name="Date début contact" nullable="True" physicalName="
          <udfs>
            <udf description="" />
          </udfs>
        </column>
      </table>
      <table alias="" name="INTERVENANT" physicalName="" superTable="">
        <column alias="" foreign="False" name="Identifiant intervenant" nullable="False" physicalNa
        <column alias="" foreign="False" name="Nom intervenant" nullable="True" physicalName="" pr
          <udfs>
            <udf description="" />
            <udf precision="" />
          </udfs>
        </column>
      </table>
    </tables>
    <relations>
      <relation LogicalName="" name="USAGER-CONTACT">
        <r0 cardianlity="0..N" multiplicity="MANY" name="USAGER" />
        <r1 cardianlity="1..1" multiplicity="EXACTLY_ONE" name="CONTACT" />
      </relation>
    </relations>
  </datamodel>
  <linkModels>
    <linkModel destination="Projet Client - Préposé" name="Client Liens" source="Modèle Conceptuel
      <link alias="1" destinationText="PRÉPOSÉ. Numéro préposé" name="Identifiant intervenant -&gt;
        <sourceCol name="Identifiant intervenant" />
        <destinationCol name="Numéro préposé" />
      </link>
    </linkModel>
  </linkModels>
</project>
```

4. Améliorer l'application Modelibra afin de mieux valider le modèle.
5. La dernière et plus importante étape était de faire la conception et la réalisation d'un prototypeur basé sur les vues de modèle.

Dans cette étape nous avons enrichi notre méta-modèle. Au début, nous n'étions pas sûr de ce que nous devrions garder, et nous avons fait une simplification de la structure de méta-modèle d'OMS. Il faut mentionner qu'OMS compte sur plusieurs niveaux de méta-modélisation, ceci parce qu'OMS supporte plusieurs approches de modélisation de même que l'interaction directe avec plusieurs fournisseurs de basse de données.

Après un certain temps, on a compris que Modelibra avait son propre chemin d'évolution et ne partageait pas les mêmes objectives du prototypeur, donc, nous avons mis des efforts pour avoir une interface commune et profiter des points forts de chacun de ces développements, et nous avons changé le format d'exportation d'OMS ainsi que l'outil d'importation pour les rendre compatibles avec le format exportation de Modelibra.

A titre d'illustration voici (4.2) la dernière version du modèle XML généré à partir de Open ModelSphere compatible avec l'exportation de modèles de Modelibra.

La compatibilité de formats exportation est un premier pas, mais il existe déjà des formats standard pour exportation des modèles. Il existe le format XMI[116] (XML Metadata Interchange - spécification OMG)) mais il n'est pas nécessaire pour le moment.

4.2.3 SoftMachine

Softmachine était un outil développé par l'auteur, Dario Gomez, à partir de 1998 ; le principe de cet outil était l'exécution des modèles à partir d'une série d'objets développés sur technologies MS, concrètement Visual Basic v06. Il a été développé selon un modèle d'architecture client-serveur. L'idée de base y est la création de vues (PCL Pseudo Concept Link) d'une table et ajouter une série de caractéristiques supplémentaires, principalement :

- Dérivées des champs de la table à laquelle faisait référence [4].
- De permettre les recherches sur la table de base et les tables de référence et afficher l'information sur une liste (tableau).
- De parcourir en tant que maître l'information relie des tables enfant (détailles) [5].

La création de la vue se fait en sélectionnant un ou plusieurs champs appartenant à la table de base, et aussi un ou plusieurs champs provenant des tables de référence (parents). La dérivation des champs se produit de deux manières : "par copie", dans lequel la table de base contient un champ (modifiable) qui reçoit une copie des champs de la table référencée et "par référence" comme un champ de seule lecture.

4. Les tableaux de référence correspondent aux clés étrangères du tableau de base.
5. Un tables « enfant » c'est celle qu'utilise table de base comme clés étrangères.

Algorithme 4.2 Exemple de document d'extraction de OMS V02 compatible avec Modelibra

```
<?xml version="1.0" encoding="utf-8"?>
<domains>
  <domain>
    <code>AI RSSSS mccd v7</code>
    <origin>OpenModelSphere 3.2</origin>
    <udpDefinitions>
      <udpDefinition>
        <code>DOCUMENT DE R/RENCE</code>
        <baseType>String</baseType>
        <alias>DOCUMENTDEREFERENCE</alias>
      </udpDefinition>
    </udpDefinitions>
    <models>
      <model>
        <idModel>3</idModel>
        <idRef>2</idRef>
        <code>Vue locale - FiTQ</code>
        <concepts>
          <concept>
            <code>Code des pays selon ISO</code>
            <properties>
              <property>
                <code>Code pays</code>
                <isNullable>False</isNullable>
                <isUnique>True</isUnique>
              </property>
            </properties>
            <foreigns />
          </concept>
          <concept>
            <code>REGISTRE DES DS</code>
            <properties>
              <property>
                <code>Code du registre des ddds</code>
                <isNullable>False</isNullable>
                <isUnique>True</isUnique>
              </property>
            </properties>
            <foreigns />
          </concept>
        </concepts>
      </model>
    </models>
  </domain>
</domains>
```

Un des points forts de ce modèle est de permettre la navigation en utilisant tous les chemins de relations existants dans la base de données. Malheureusement, ce projet a été abandonné. Mais les idées de base ont guidé tout le développement du prototypeur.

4.3 Choisir un outil pour le développement

Finalement, dans l'univers du logiciel libre, nous avons constaté que pour le développement Web, Java n'était pas la seule option, et que bien des réalisations innovatrices n'étaient pas développées directement sur ce langage. Il n'existe pas de technologie ultime permettant de tout faire de manière optimale. Chaque outil, créé pour le développement, répond initialement à une demande spécifique à laquelle aucune autre solution préexistante ne semblait répondre convenablement. Particulièrement quand on se pose la question d'un développement Web il y a deux mots qui dominent CMS ou Framework.

4.3.1 Qu'est-ce qu'un CMS ?

Les CMS (Content Management System) sont des outils / logiciels offrant des fonctionnalités livrées clé en main et rapidement installables. Ils permettent de développer un site vite et à moindre coût. Le peu d'efforts nécessaires à leur mise en place les rend accessibles sans avoir besoin de connaissances techniques poussées. En revanche, leurs avantages sont aussi leurs inconvénients. Le fait d'avoir des fonctionnalités déjà développées les rend plus difficilement modifiables.

Les CMS les plus populaires sont [6] :

- Wordpress, Drupal, Joomla, typo3 (PHP)
- Django CMS, Merengue, Mezzanine, Plone (Python)

4.3.2 Qu'est-ce qu'un Framework ?

La définition du terme anglais « Framework » veut littéralement dire « cadre de travail ». L'intérêt premier d'un framework, c'est d'abord de gérer l'architecture d'une application en délimitant les normes de développement et en séparant deux choses essentielles : le code métier et le reste. Les frameworks sont constitués par un ensemble de librairies correspondant aux tâches les plus répétitives du développement. De cette manière, un développeur peut se consacrer exclusivement au code métier de l'application sur laquelle il travaille, sans jamais toucher aux librairies ou alors occasionnellement en ajoutant des extensions à un package qui ne fournit pas tout à fait le résultat attendu, mais sans toucher bien entendu à l'original.

6. cette liste n'est pas exhaustive

Le framework est en quelque sorte un intermédiaire entre le code métier et les librairies. Leur architecture est bien plus flexible que celle des CMS et leurs fonctionnalités plus avancées. Les frameworks ont eu un rôle éducationnel indéniable en apportant au monde du développement des pratiques visant à en améliorer la qualité du développement.

Les Frameworks que nous avons considéré choisis parmi les plus populaires sont :

- Django (Python)
- Symfony, Zend (PHP)
- Ruby on Rail (Ruby)
- Spring, Eclipse RAP (Java)

4.3.3 Quoi choisir ?

Étant donné que notre besoin n'est pas l'administration du contenu. On est parti à la recherche d'un framework. Il y a quelques années encore il n'était pas possible de trouver un framework web, mais aujourd'hui le nombre de framework web est quasi indénombrable.

Nous avons débuté une étape d'exploration avec les « frameworks » qui nous semblaient les plus avancés et qui avait une communauté active. Voici une liste des produits que nous avons essayé. On a fait de petits projets pilotes qui nous ont permis de prendre une décision.

Le framework retenu est Python-Django. Les critères sont très bien résume par Daly (2007) « *Django advocates a "model-centric" approach to development, in which all the essential fields and behaviors of the data (and thus much of the behavior of the application itself) are part of the model. ... the model is meant to be designed primarily in Python code. The database schema and data maintenance process are handled by Django based on that model. This is the core of Django's adherence to the "don't repeat yourself" principle: the model is described in one place, and the messy details of persisting it to a database are hidden* »[111]

4.3.4 Django Admin

Après avoir choisi Django, nous avons commencé à explorer l'Admin[7]. Une des composantes les plus puissantes de Django est l'interface d'administration automatique. Elle lit les méta-données dans le modèle pour fournir une interface simple mais puissante, prête pour commencer à être utilisée immédiatement pour ajouter du contenu à la base de données.

Il semble similaire à notre objectif, et le code source était disponible. Pendant plusieurs mois nous avons travaillé sur l'Admin pour le modifier afin d'ajouter des fonctionnalités nécessaires du prototypeur. L'étude du code Admin et parfois même le code de base Django, nous a permis de comprendre son fonctionnement. Maintenant nous n'utilisons plus les modifications que nous avions faites, parce que nous limitons l'utilisation de Django à la partie BackEnd.

7. https ://docs.djangoproject.com/en/dev/ref/contrib/admin/

4.3.5 Projet TCO

Nous ne pouvons pas faire des choses génériques sans avoir travaillé avec un exemple. La généralisation d'un concept est une extension de critères moins spécifiques. Il faut maîtriser le concept particulier avant de s'aventurer dans un niveau d'abstraction de généralisation. Le processus de vérification est nécessaire pour déterminer si une généralisation est valable pour une situation donnée. Donc, nous avons choisi un projet de test à développer pour créer un scénario de preuves pour la création du prototype. Nous l'avons fait directement à la main de façon à identifier ensuite les patrons de généralisation et pour obtenir le même résultat. Ce projet portait sur le contrôle de coût total de possession (TCO Total Cost Ownership) d'un logiciel.

Pour l'exécution du projet TCO nous avons utilisé Django Admin. Nous avons eu besoin d'ajouter des fonctionnalités dans le côté client que l'admin ne fournissait pas. En fait, nous avons continué à développer un site Django sans l'utilisation de l'Admin. Mais le développement Web est long et pénible. Nous avions 14 entités de base dans le modèle conceptuel de données, et pour chacune il fallait créer au moins deux templates, le premier pour la liste et le deuxième pour l'édition. Sans compter les rapports et d'autres fonctionnalités nécessaires.

Nous avons évalué la possibilité de modifier les templates de l'admin afin de pouvoir ajouter une petite partie des fonctionnalités manquantes, par exemple changer un filtre d'un combo dynamiquement en fonction d'une autre valeur déjà modifie. Par exemple : dans le cas de Logiciel et famille logiciel, on devrait sélectionner pour les équivalences un logiciel appartenant a la même famille, c'est le cas plus simple, mais cette fonctionnalité n'est pas considéré de façon native a l'intérieur de Admin Donc, il fallait soit modifier les templates d'Admin, soit faire de templates indépendants.

L'expérience de faire les pages web, n'était pas très agréable, notre stagiaire a utilisé deux mois (à vrai dire pas à temps plein) pour créer les listes, et une édition rudimentaire. Voici un aperçu de la page d'accueil (fig 4.1).

4.4 Ce qu'on retient

4.4.1 ExtJs

Sencha ExtJs est présenté comme la plus puissante plate forme de développement des applications de bureau [8] . ExtJs offre la possibilité d'une interface applicative riche et aussi puissante que n'importe quel outil de client lourd. Il permet la compatibilité cross-browser, une architecture MVC, des objets graphiques et une interface utilisateur moderne. Mais il ne s'occupe pas de la gestion de la persistance de données.

8. http ://www.sencha.com/products/extjs

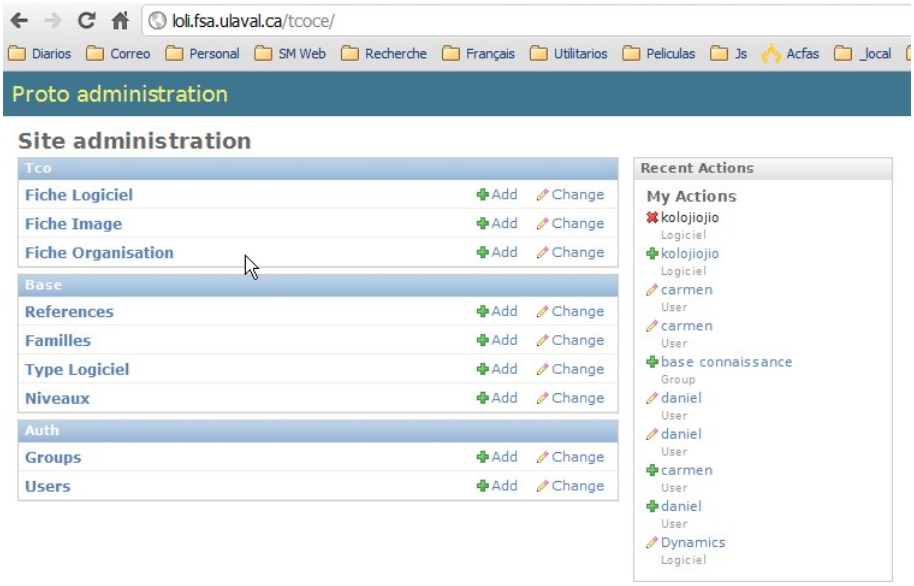


FIGURE 4.1: Projet TCO

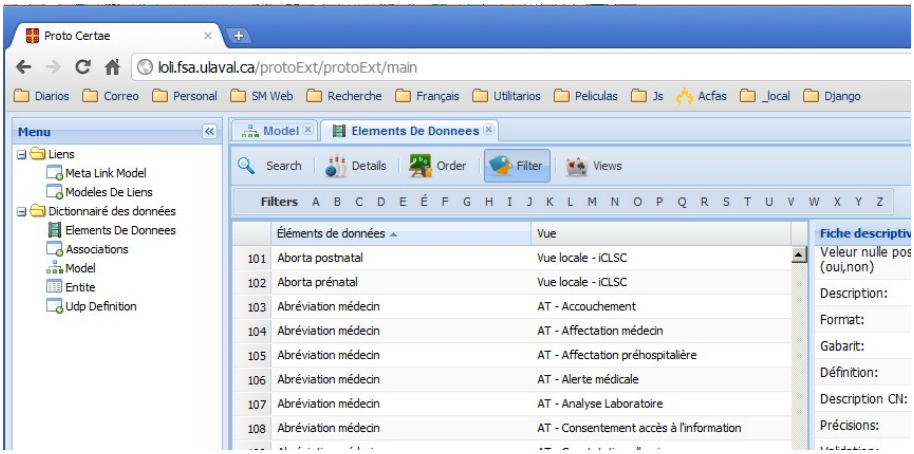


FIGURE 4.2: Dictionnaire MSSS

ExtJs utilise un ensemble de techniques de développement web du côté client pour créer des applications Web asynchrones : JavaScript (souvent abrégé JS) est un langage de programmation de scripts principalement utilisé dans les pages web interactives, mais aussi côté serveur. HTML et CSS peuvent être utilisés en combinaison pour construire des pages. Le DOM est accessible avec JavaScript pour afficher de façon dynamique, et permettre à l'utilisateur d'interagir avec l'information présentée. JavaScript et l'objet XMLHttpRequest fournissent une méthode pour échanger des données de façon asynchrone entre le navigateur et le serveur afin d'éviter la recharge totale de la page. Ajax (acronyme d'Asynchronous JavaScript and XML), permet aux applications web de communiquer avec un serveur pour envoyer et récupérer des données de manière asynchrone sans interférer avec l'affichage et le comportement de la page existante. Malgré son nom, l'utilisation de XML n'est pas nécessaire, JSON est souvent utilisé

à sa place, ce que nous avons choisi.

La courbe d'apprentissage de ExtJs est assez rapide, on dispose de nombreux exemples avec des cas pratiques. Cependant la documentation concerne l'utilisation "normale" du Framework pour faire une application spécifique. Le prototypeur demande plus car il requiert une approche dynamique.

Le fait qu'il est open source nous a permis d'aller étudier la façon le fonctionnement interne et d'apprendre comment implémenter les fonctionnalités nécessaires pour le prototypeur.

Il y a des risques a utiliser des caractéristiques non documentées, par exemple, dans un changement de version, les programmeurs peuvent conserver l'interface, mais changer ce qui n'était pas officiel. Ceci fait qu'on doit toujours tester en profondeur n'importe quel changement de version, même si c'est mineur.

4.4.2 Django

Django est capable de gérer de sites comme instagram, mozilla ou pinteres. Pourtant son orientation de client HTML de base, ne fournit pas le dynamisme d'un application transactionnelle de bureau. Un site web basé sur HTML doit fournir une page HTML pour chaque requête, cette page peut être générée de façon automatique, mais le serveur et le client sont forcé de traiter une nouvelle page à chaque occasion.

- Un des points forts de Django c'est son ORM (acronyme d'object-relational mapping) qui est une technique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre la définition de classes et la base de données. On pourrait dire que c'est le pont « entre monde objet et monde relationnel » [9]. Le ORM de Django permet de définir les modèles de données entièrement comme des classes Python.
- Django permet aussi l'interprétation des URL et l'assignation aux méthodes accessibles par moyen des services REST (REpresentational State Transfer) qui est un style d'architecture adapté pour les systèmes hypermédia distribués. Dans REST les responsabilités sont séparées entre le client et le serveur. L'interface utilisateur est séparée de celle du traitement des données. Cela permet à chaque côté d'évoluer indépendamment. Autre caractéristique de REST c'est que chaque requête du client vers le serveur doit contenir toute l'information nécessaire pour permettre au serveur de comprendre la requête, sans avoir à dépendre d'un contexte conservé sur le serveur. Cela libère de nombreuses interactions entre le client et le serveur. Chaque requête du client est aussi appelée un message. Et chaque point d'entrée sur le serveur est aussi appelé un service. REST c'est le point de rencontre de Django avec ExtJs.

9. Le modèle objet et le modèle relationnel ne sont pas totalement compatibles, ce que Scott Ambler appelle « désadaptation d'impédance ».

- Django permet aussi la définition des « Middleware » des systèmes de crochets (hook) de «requête / réponse" de bas niveau pour modifier globalement l'entrée et / ou sortie de messages.
- Django possède un support complet pour les applications mufti-langues, permettant de spécifier les chaînes de traduction et de fournir des crochets (hook) pour une fonctionnalité spécifique à la langue.

Nous n'utilisons le système de gabarits (Django template) que pour fournir un page de base ou héberger le code dynamique de ExtJs.

Chapitre 5

Conception et architecture

5.1 Délimitation de la portée

5.1.1 La relation entre le réel perçu et les données

Tout SI repose sur une architecture d'information, c'est avant tout un contenu au sujet d'une réalité, le réel perçu, que l'on peut exprimer par un modèle conceptuel données MCD. S'il est normalisé,[1] en s'appuyant sur les identifiants connus par les utilisateurs du SI, il renferme implicitement toutes les navigations permises par les règles d'intégrité (connectivités ou cardinalités). Dans ce chapitre nous présentons la génération des applications à partir d'un modèle réalisé sous forme de métadonnées mise en œuvre dans le prototypeur. Pour cela nous avons développé une structure d'accueil assez riche pour supporter la définition complète d'un MCD. Le prototype doit permettre les quatre opérations d'édition habituelles CRUD (Create, Read, Update, Delete) et toutes les navigations permises par le MCD afin de maintenir l'intégrité des données. Nous partons du principe que la structure des données et le comportement dans les différentes interfaces utilisateur peuvent être fabriquées à partir de métadonnées. L'objectif final est de produire un prototype fonctionnel.

5.1.2 Prototypes vs production

> « A prototype produces "running" software and the production development produces "working" software." » (Hantos 2000 :3)[86]

Un belle métaphore utilisée par Poppendieck dans son livre « Lean software development : an agile toolkit » [136] illustre que le développement est tout à fait différent de la production. Il propose de penser au développement comme à la création d'une recette et à la production

1. au sens des 3 formes normales de la théorie des bases de données relationnelles

comme le suivi de la recette. Ce sont des activités très différentes, et elles doivent être effectuées avec des approches différentes. Développer une recette est un processus d'apprentissage impliquant des essais et des erreurs, les chefs les plus experts ne s'attendent pas à avoir un plat réussi à la première tentative. L'élaboration d'un système est un processus de conception progressif facilité pour l'élaboration de prototypes. Mais, il faut faire attention, le prototype fait abstraction de fonctionnalités comme la performance, les accès concurrents, la sécurité dans tous ses aspects. ... qui bien qu'indispensables ne contribuent pas à la compréhension du fonctionnement du système. Le but d'un prototype est de concrétiser les conséquences, au plan fonctionnel, de la représentation du réel perçu en ce qui concerne les données[2] afin que les utilisateurs puissent vérifier l'intégrité, c'est-à-dire la cohérence et la complétude (au moins avec la connaissance actuelle) de la solution proposée.[44]

Le prototypeur généré dans le cadre de ce mémoire n'est pas adapté à tous les types d'applications. Pour le moment, il ne convient qu'aux petites et moyens applications axées sur les données. Ce projet est cependant une étape vers un outil capable d'administrer tous les artefacts de l'architecture d'entreprise. Étant donné sa souplesse, il pourrait aussi satisfaire des besoins spécifiques, en adaptant et en ajoutant des fonctionnalités spécifiques selon l'objectif visé.

5.1.3 Exigences minimales

Un des plus grands apports dans la réalisation des systèmes informatiques au cours des dernières années réside dans la puissance des interfaces utilisateur avec les navigateurs web qui permettent d'exploiter des architecture technologiques en client léger. Ils apportent un grand nombre d'avantages, entre autres : aucun logiciel client à installer et un accès universel. On a choisi un style d'architecture de séparation en couches, on isole la couche d'interface utilisateur qu'on appelle, suivant l'usage, FrontEnd et la couche de traitement qu'on appelle BackEnd.

Nous avons retenu les exigences suivantes pour ces deux couches et leur articulation :

1. Client Web (FrontEnd) :

    a) Le client doit être dynamique, en tirant parti des caractéristiques essentielles connues sous le nom de Web 2.0. Il s'agit en particulier des techniques d'application Internet riches telles qu'AJAX pour améliorer ce qu'il est convenu d'appeler l'expérience utilisateur. Par exemple, elles organisent les échanges des informations entre le client et le serveur afin de mettre à jour une partie seulement du contenu de la page affichée, sans rafraîchir la page entière, ceci permet un dialogue à distance très sophistiqué et rapide.

2. telle que nous l'avons définie dans le ch 1

b) Spécification d'interface : Le module de spécification d'interface est fondamental dans la réalisation d'un prototype, il réalise la façon dont l'utilisateur interagit avec le système informatique.

2. Fournisseur de services Web (BackEnd) :

a) nous recourrons à ORM (object-relational mapping) : c'est une technologie pour la définition d'un modèle de domaine respectant la façon dont l'organisation pense à ses données exprimées dans le MCD, plutôt qu'uniquement la façon dont les données sont manipulées et stockées. L'ORM assure également une indépendance avec l'infrastructure technologique ou le fournisseur des systèmes de gestion des bases de données.

b) Traitement et de la persistance sur une base de donnés exploitant les connaissances apportées par la technologie de programmation objet

5.2 Formalisme de modélisation objet-relationnel

5.2.1 Modélisation du Domaine

> "UML 2.0 lacks both a reference implementation and a human-readable semantic account to provide an operational semantics, so it' s difficult to interpret and correctly implement UML model transformation tools." (Thomas 2004:16)
>
> "If you feel constrained by the capabilities of UML, you will often have to leave out the most crucial part of the model because it is some rule that doesn't fit into a box and line diagram. And, of course, a code generator cannot make use of those textual annotations." (Evans 2004:32)

La modélisation "Entité-Relation (ER)" sous ses diverses variantes a été développée afin d'aider les concepteurs à visualiser puis à implémenter des bases de données relationnelles sous forme de tables avec leurs liens. Cette technique a été une étape importante qui a été intégrée aux méthodes de conception des SI dès le milieu des années 70, ce qui fait que l'on dispose aujourd'hui d'une base de traitement éprouvée en ce qui concerne les données (telles que nous les avons introduites dans le premier chapitre).

Par la suite, le langage de modélisation unifié (UML) a été introduit afin d'accélérer, simplifier et clarifier la spécification et la programmation des systèmes informatiques, il s'agissait de mettre en forme réutilisables les connaissances que les programmeurs ont progressivement formalisées pour « industrialiser » leur travail. Cette mise en forme est conçue pour respecter les exigences des utilisateurs du SI. Des composantes d'UML sont dérivées de la modélisation ER. L'ORM (Object/Relational Mapping) fournit un mécanisme orienté objet pour manipuler

les données persistantes[3] à long terme dans une base de données relationnelle en assurant l'intégrité référentielle. L'ORM encourage l'utilisation des modèles et des contraintes dans un contexte objet avec une persistance relationnelle. L'ORM est l'outil parfait pour modéliser un domaine (le réel perçu) parce qu'il fait le lien entre la conception objet et la manipulation relationnelle des données.

UML offre plusieurs choix pour représenter les concepts comme l'optionalité, la cardinalité et la multiplicité dans les diagrammes de classe. Nous pensons qu'il est important d'avoir un formalisme claire et non ambigu. Donc, nous avons décidé d'adapter un formalisme plus en harmonie avec les outils que nous utilisons dans notre travail. En suivant l'idée de l'ORM, on considère qu'il est important de définir une type de modélisation qui fasse le lien « objet-relationnel OR » pour assurer une application directe entre la définition de domaine (les données établies par l'analyse du SI) et la création des modèles ORM qui exploitent la connaissance apportée par les pratiques OO[4].

Nous présentons ici la définition d'une structure de métadonnées capable d'emmagasiner les modèles et de définir les spécifications nécessaires pour l'interprétation automatique du prototype :

- les composantes des métadonnées pour emmagasiner les modèles qui sont directement inspirées des modèles de données :
  1. le « Projet » (« project »)[5] qui correspond à un SI
  2. le «Modèle" ("model") qui correspond à une unité d'intervention sur le réel perçu
  3. le « Concept » (« concept ») qui est l'unité de perception fondamentale, il permet l'identification d'une chose[6] constituant le réel perçu
  4. la « Propriété» (« Property ») qui porte les valeurs décrivant un concept[7] et ses « attributes »
  5. la « Relation » (Relationship) qui permet de lier les concepts.
- les composantes des métadonnées pour manipuler les données qui s'appuient sur les relations qui sont établies entre les concepts (les couples clé primaire, clé étrangères du modèle conceptuel relationnel) sous des formes adaptées à la programmation objet et qui s'appuient sur les notions d'identité et de référence :
  1. agrégation

3. L'usage du mot persistant est significatif, s'il a un sens pour le programmeur qui dans l'organisation et la réalisation de son code doit définir de nombreuses variables temporaires, il est surprenant pour les utilisateurs du SI pour qui les données doivent par définition être mémorisées.
4. OO : Object Oriented
5. Dans le code, nous avons utilisé des noms anglais, qui se retrouvent dans les diagrammes alimentant le code. Ceci pour des raisons de communication avec les communautés de logiciel libre. Nous utiliserons donc les noms anglais dans les diagrammes et les noms français dans le texte les commentant ou utilisant.
6. On aimerait dire ici « objet », mais nous l'évitons pour ne pas créer de confusion avec le sens OO.
7. Ce mot était utilisé dans Merise au départ, et il correspond bien à ce que nous entendons ici

2. composition
3. héritage

– les vues d'utilisateur basées sur l'implémentation d'un modèle de spécification d'interface (MSI) dérivé de la méthode Datarun. Dans la méthode Datarun on définit des blocs qui sont basés sur les entités (ou tables) du modèle conceptuel et complétés par des données d'autres tables en exploitant les trois opérateurs relationnels, projection, sélection et jointure. Puis ces blocs sont agencés à l'aide des mêmes opérateurs, essentiellement la jointure, pour former une unité d'interface appelée MSI. Ceci peut être spécifier graphiquement sous la forme d'un modèle de données, à l'aide de dérivées par copie, ou sous forme syntaxique par le biais d'une convention relative aux opérateurs. Du point de la réalisation du prototype, le MSI est réalisé par une vue comme une autre, car les vues sont progressivement construites à partir d'un concept. Nous appelons donc ici MSI une vue qu'elle soit mise à la disposition de l'utilisateur en tant que module d'interface ou simplement utilisée comme composante intermédiaire car du point de vue de la programmation on ne fait pas cette distinction importante pour les utilisateurs.

1. un MSI est basé sur un concept
2. dans un MSI seul le concept de base peut être modifié (opérateurs CRUD), les propriétés qui n'appartiennent pas au concept de base ne sont pas modifiables.
3. un MSI peut contenir des propriétés appartenant au concept de base, ou des propriétés dérivées de ses références. Une propriété est considérée dérivée par référence quand elle n'existe pas dans le concept de base mais dans un concept référé [8].
4. un MSI doit définir la navigation vers les concepts liés en respectant les relations établies entre eux.
5. un MSI peut être un point d'accès dans l'application (option de menu utilisateur)

5.2.2 Les composantes issues du réel perçu

Concept, Propriété et attribut

Le concept est la principale construction de la modélisation objet-relationnel. Il correspond à une classe dans OO, une table dans une base de données relationnelle et une entité dans un modèle Entité-Relation.

Le première section contient le nom du concept, la deuxième section les propriétés avec certains de leurs attributs, en suivant une convention généralement acceptée décrite ci-après (5.2.2).

8. Cette différence entre copie et référence est importante pour le programmeur alors qu'elle n'est pas perceptible pour l'utilisateur qui verra toujours la propriété dans son interface.

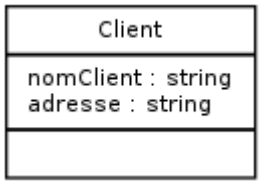


FIGURE 5.1: Un concept et ses propriétés

Les propriétés sont les éléments de données qui portent les valeurs, en fait c'est ce qui intéresse vraiment les utilisateurs du SI. On les représente graphiquement dans la deuxième partie du concept.

Il est commode pour le programmeur de pouvoir, même graphiquement, spécifier certains attributs des propriétés comme les attributs de codage (string dans l'exemple), attributs qui importent souvent peu pour les utilisateurs du SI. Ce qui allait de soi dans le réel perçu (le codage) doit être explicite dans le monde abstrait isolé d'un contexte général de connaissances qu'est l'ordinateur.

Les concepts doivent être identifiés, comme c'est là un point central de l'articulation entre le modèle du domaine et le modèle objet-relationnel nous présentons la problématique de l'identification ci-après sachant que les clés primaires et les clés étrangères sont les identifiants utilisés dans le réel perçu qui doivent être visibles dans les interfaces (5.2.3).

Convention pour la formation des noms

On dit parfois qu'il y a autant de convention pour former des noms [9] valides qu'il y a des concepteurs. Nous essayons généralement de nous conformer aux « philosophies » des outils quand il s'agit de nommer les éléments de conception. Cependant, comme nous travaillons avec plusieurs technologies que ne sont pas nécessairement d'accord dans la façon de nommer les éléments (classes, propriétés, méthodes, etc). D'un côté Django fait des recommandations[2] et s'appuie sur les recommandations de Python[10]. De l'autre cote ExtJs propose ses standards[145]. Pour régler la situation nous en sommes venu à préférer les notationsCamel[15]. À notre avis, c'est plus lisible et il y a un peu moins de caractères que la notation avec des trait de soulignement. À vrai dire, la seule chose importante est de garder la même convention de formation des noms dans le projet.

La convention que nous utilisons est la suivante :

- Les noms des concepts utilisent « UpperCamelCase ou CamelCase », par exemple :
  - Client
  - LigneFacture

9. coder : au sens large du terme, par exemple nommer les éléments de conception

- Les nom des propriétés utilisent « lowerCamelCase ou camelCase », par exemple :
  - adresse
  - nomClient

Les verbes dans les noms devraient être réservés aux actions en général (méthodes, procédures, taches, etc ..)

Il est possible d'utiliser :

- caractères accentués, le système les utilisera tel quel pour les libellés et les changera automatiquement par la lettre la plus proche pour les noms internes.
- espaces, le système changera la lettre après l'espace pour une majuscule.

Il est déconseillée d'utiliser :

- seulement des chiffres.
- des caractères spéciaux tel que les traits de soulignement et les points. [10]
- des noms avec moins de trois lettres.

5.2.3 Les composantes apportées par les connaissances informatique (approche objet)

Ces composantes sont pertinentes dans le sens où elles identifient un sous-ensemble de comportements du point de vue de leur exigence commune dans les programmes, notamment en ce qui concerne les relations entre entités. C'est une manière d'incorporer une connaissance générale dans l'architecture des logiciels qui vient compléter par exemple celle qui était implicite avec les connectivités ou cardinalités du modèle conceptuel de données. Remarquons que les relations nommées dans le MCD portent une signification précise [11], mais ce n'est pas cette signification qui est utilisées pour forger ces composantes.

Ainsi, un concept peut contenir des propriétés qui font référence à d'autres concepts, c'est ce que veut dire un connecteur relationnel mis en œuvre par une clé étrangère. La définition d'un référence implique la création d'une propriété avec un attribut type de donnée équivalent au concept référencé, c'est aussi l'équivalent d'un objet qui fait partie d'un autre objet.

Clé primaire, Clé étrangère, Clé sémantique

Nous considérons, au plan de la réalisation du code, qu'un concept (donc dans le monde des représentations qui sont des abstractions) a toujours implicitement par défaut une identité qui

10. La notation objet de certains langages et frameworks utilisent « . » et « _ » pour explorer les objets composants ou nommer les critères de recherche.

11. Si elle est facilement utilisable par les utilisateurs du SI, elle ne l'est pas dans l'état actuel pour la production du code à partir des métadonnées

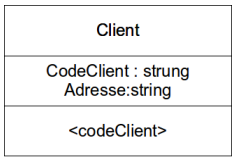


FIGURE 5.2: Un concept, ses propriétés et la clé sémantique

désigne de manière unique chacune de ses instances, sans avoir besoin d'un autre qualificatif. L'identité est obligatoire, mais n'est pas spécifiée dans le modèle, elle est implicite mais prise en charge par le code sous-jacent. [12]

La clé sémantique, est matérialisée par une ou plusieurs propriétés qui permettent à l'utilisateur de reconnaître l'information. La sémantique consiste à ce qu'une relation bi-univoque soit établie entre l'existence d'une chose dans le réel perçu et sa représentation du point de vue des utilisateurs. La clé sémantique nous sert à réaliser l'identification, mais n'est pas l'identité. On considère que la « clé sémantique » connue par les utilisateurs est un élément important fondamental du modèle.

Pour des raisons techniques il se trouve que dans l'ordinateur la clé effectivement utilisée doit en être dérivée. En effet, ce qui va implicitement de soi dans le monde réel des utilisateurs du SI, ne va plus de soi avec les contraintes imposées au programmeur du fait de l'abstraction et de l'isolement du programme et de ses données de l'ensemble des connaissances dont les utilisateurs font implicitement usage.

Une métaphore, souvent utilisée par Jean-Louis LeMoigneinline false illustre bien cette problématique : « le couteau de Jean », tout le monde s'accorde à reconnaître le couteau de Jean même si la lame puis le manche ont été changés, l'Identifiant couteau de Jean est parfaitement utilisable au sein des proches de Jean, mais il n'en va pas de même avec la représentation informatique : c'est cette problématique que doit prendre en charge la modélisation OR [13].

Par exemple (figure)5.2 : Le concept "Client" est composé du : codeClient, nomClient et adresse. "codeClient" est défini comme clé sémantique afin de manipuler les représentations des "Clients". Cette clé, connue comme clé primaire dans le modèle de domaine, est celle que les utilisateurs utilisent pour faire la relation entre la chose "Client" et sa représentation, d'où son qualificatif de clé sémantique. Attention le "codeClient" n'est pas une clé informatique satisfaisante du point de vue du code qui fonctionne suivant un principe de référence dénué de signification autre que permettre de passer de manière fiable d'une zone de la mémoire à une autre.

Il est aussi possible de combiner plusieurs propriétés pour former un clé sémantique. Par exemple si le concept "Client" est composé du "nom" , "prénom", les utilisateurs, si cela est

12. Dans la construction interne (n'oublions pas que nous faisons une application sur un modèle relationnel) l'identité peut être un nombre entier arbitraire ou un identifiant unique (UUID ou GUID).
13. OR : objet-relationnel

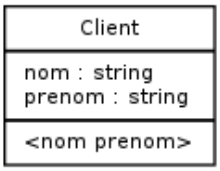


FIGURE 5.3: Clé sémantique composée

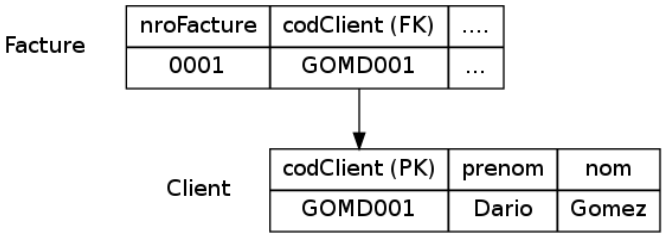


FIGURE 5.4: Utilisation des clés sémantiques dans le modèle ER

acceptable (combinaison unique) dans leur réel perçu, de définir "nom prenom" [14] comme clé sémantique.

Si l'existence d'une clé sémantique est obligatoire, il y a des cas où il n'est pas nécessaire de l'indiquer explicitement, si elle peut être déduite de la structure des relations dans le modèle.

Voici une illustration de cette problématique entre la modélisation ER (figure 5.4) et la modélisation OR (figure 5.5) avec un exemple de facture et de client, dans le modèle de domaine les relations sont concrétisées par les clés sémantiques sous la forme de couple clé primaire - clé étrangère :

Voici l'exemple avec la modélisation ER traditionnel (figure 5.4) :

- Le client est identifié par un identifiant sémantique, codeClient est la clé primaire du client dans l'entité Client
- La facture a une clé étrangère « codeClient » qui établit la relation avec un client »

Voici l'exemple avec la modélisation OR qui transforme les clés sémantiques en références, c'est à dire en pointeur, on doit passer du point de vue sémantique de « cette chose est ça » à « cette chose est là » (figure 5.5) :

- le client est identifié par un id (donné automatiquement lors de sa création) ;
- la facture a une propriété de type « client » qui pointe vers le concept « Client ». la facture pointe vers un Id automatique que l'utilisateur n'a pas besoin de connaître.
- Le Client a un clé sémantique « codClient + nomClient », c'est un règle (connaissance) de représentation définie par l'utilisateur. Le code considère que la clé sémantique est une règle, elle ne pointe pas directement vers l'emplacement connu du code.

14. Pour des raisons techniques, nous n'utilisons pas de caractère accentué dans les diagrammes et le code

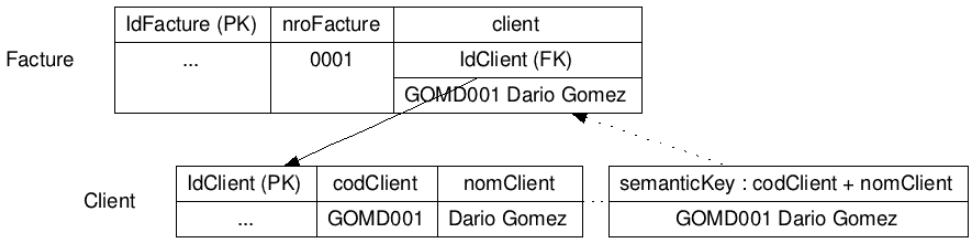


FIGURE 5.5: Utilisation des clés sémantiques dans le modèle OR

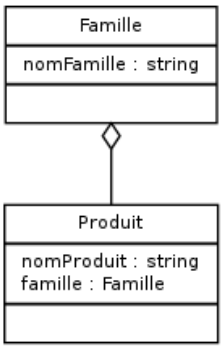


FIGURE 5.6: Référence

- Représentation interne (a représentation du client dans la facture est la clé sémantique. De plus la facture peut invoquer n'importe quelle propriété du client au moyen de la notation objet) :
  - >>> myFacture = Facture.get(nroFacture = 0001)
  - >>> print myFacture.client , « ; », myFacture.client.codClient
  - >>> « GOMD001 Dario Gomez » ; « GOMD001 »

Relation et références

Une référence correspond à une relation entre les concepts, elle ne peut pas comporter plus de deux concepts (limitation apportée elle aussi par les modèles conceptuels relationnels de données). Elle est représentée par une ligne entre les deux concepts. Si elle a un nom, il peut être écrit dessus, mais ce nom n'est pas utilisé par le code sous-jacent. Cette dernière remarque est importante, dans le modèle objet-relationnel, ce qui importe est le comportement général associé à la relation.

Dans cet exemple, la famille a une collection des produits. Le produit appartient à une famille, la famille peut ainsi être considéré une propriété du produit. On dit que "Famille" est un parent de "Produit", et vice versa un "Produit est un enfant de "Famille". La relation implique un hiérarchie des concepts et on peut naviguer suivant les deux directions des relations avec des

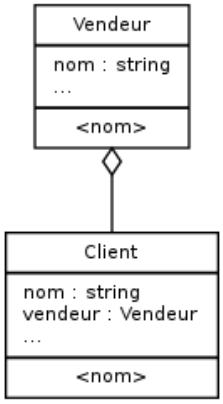


FIGURE 5.7: Relation de Agrégation

résultats différents. On peut consulter *un* parent ou consulter la *liste* des enfants. Cette notion n'est pas absolue, un concept peut être parent dans une référence (relation) et un enfant dans une autre référence (relation). Le parent peut être un objet étendu (un agrégat) qui peut alors être considéré comme une unité bien que physiquement composé de plusieurs objets plus petits.

Si on a besoin de faire un liste de produits, le nom de produit est évident, mais comment faire pour présenter la famille. On a besoin d'une propriété (ou plusieurs) qui représente le contenu de chaque enregistrement, ce qui nous conduit alors à utiliser la clé sémantique.

Agrégation L'agrégation représente un couple qui peut être traité comme un ensemble (conteneur) et des composantes (contenus) regroupées dans cet ensemble. L'agrégation est utile quand un concept peut être considéré comme une collection contenant d'autres concepts et que les concepts contenus n'ont pas une forte dépendance d'existence sur le conteneur, c'est-à-dire que si le conteneur est détruit, son contenu ne l'est pas.

Le diamant vide indique que le « Vendeur » assigné (le contenant) peut disparaître et le « Client » (le contenu) ne disparaîtra pas [15]. La propriété vendeur dans le concept « Client » prend par défaut « null » [16]. On remarquera que cette notation rend plus explicite le comportement du logiciel sur les concepts que la simple utilisation des connectivités. Ceci est caractéristique du changement de perspective lorsque l'on passe de la vue des données à celle du logiciel basé sur un corpus de connaissances spécifique : le graphisme indique clairement que l'on peut manipuler dans le code le conteneur avec le contenu, il indique aussi par ailleurs que le couple conteneur-contenu peut être considéré comme un seul objet composé [17].

15. On voit bien que dans cet exemple l'agrégation est définie du point de vue de la manipulation de la référence -relation- et non d'une signification stable du point de vue des usagers du SI.

16. Cela correspond à la règle dans le parent "on delete set null" dans la base de données.

17. Le changement de formalisme repose sur le changement de perspectives impliqué par la nature de l'orientation objet

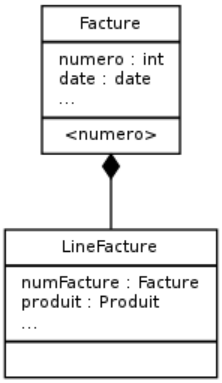


FIGURE 5.8: Relation de composition

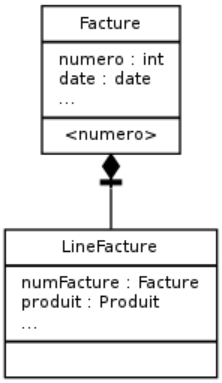


FIGURE 5.9: Relation de composition protégée

Composition La composition est une relation qui représente aussi un couple qui doit être traité comme un ensemble (conteneur) et des composantes (contenus) regroupées dans cet ensemble, mais la composition est utile quand les concepts contenus ont une forte dépendance sur les concepts conteneurs, si le conteneur est détruit, son contenu l'est aussi.

Le diamant plein indique que si la facture est effacée, les lignes de facture sont effacées en même temps. Cela correspond à la règle "on delete cascade" dans la base de données.

Composition ou Agrégation protégée Un cas spécial de relation se produit lorsqu'on le parent ne peut pas être effacé avant que tous ses enfants aient été effacées.

Le diamant (plein ou vide selon type de relation) et la barre indiquent que la facture est protégée contre l'effacement tant qu'elle contient des lignes de facture associées. Cela correspond à la règle "on delete rollback" dans la base de données.

Héritage (spécialisation/généralisation) L'héritage est une relation dans laquelle les concepts de l'élément spécialisé (concept enfant) peuvent remplacer les concepts de l'élément

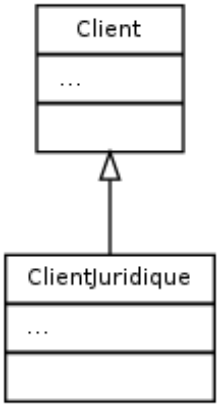


FIGURE 5.10: Relation de héritage

général (concept parent). Par exemple : Un client juridique est un sorte spécialisée du client.

L'héritage est représenté par une flèche fermée vide qui pointe sur le concept qui hérite (le parent). Un relation d'héritage implique une relation 1 :1 dans son équivalent relationnel.

5.3 MCD Modèle conceptuel de données : l'input du prototypeur

La méthode Datarun préconise de réaliser le plus tôt possible le modèle de données primaires ou modèle conceptuel de données « MCD » car toutes les tâches sont organisées autour et à partir de ce modèle de données. Ceci veut dire que si la réalisation du MCD demande une connaissance préalable des processus (souvent avec des modèles), elle précède tout diagramme de traitement de l'information impliquant le recours au SI. Les activités qui précèdent la réalisation du MCD doivent être limitées à ce qu'il est nécessaire de découvrir et de documenter pour sa réalisation. L'input du prototypeur dont l'objectif est d'aider les utilisateurs en relation avec les analystes est de structurer le SI en vue de son informatisation est donc le MCD.

Pour accueillir un MCD, il faut un structure capable de gérer la sémantique, c'est à dire assurer la relation entre les choses du réel perçu et le contenu de la base de données. Nous avons retenu une approche agile en s'appuyant sur l'existence de la plupart de ce qui est nécessaire pour atteindre l'objectif de construction rapide d'un SI flexible qui réponde adéquatement aux exigences du domaine. Cette approche est utilisée deux fois. Une première application est celle de l'usage du prototypeur pour la spécification de domaine puis la construction du SI. Lors de cette application les MSI sont progressivement élaborés en s'appuyant sur les fonctionnalités du prototypeur.

La deuxième est celle de la réalisation de l'outil de prototypage, objet de notre travail. Nous avons donc commencé suivant la décision initiale de s'appuyer sur des méta-données à définir

un dictionnaire des données (en fait il y a eu plusieurs itérations avant d'arriver au dictionnaire retenu). Ce dictionnaire doit recevoir le MCD et servir de base pour générer la base de données en passant par la couche d'abstraction de l'outil retenu pour le BackEnd (Django). Chaque spirale est composée d'un modèle, de l'implémentation et de l'évaluation de son intérêt pour la réalisation d'un prototype.

Il est important de mentionner qu'au fur et a mesure que le modèle a été développe, nous avons développe l'outil base sur le même modèle. En effet, dans une approche de « bootstrapping » adaptée à cette situation, l'outil de prototypage était construit avec lui-même : l'interface de saisie du modèle est en effet réalisée à l'aide du prototypeur (il en va bien sûr de même pour le dictionnaire, qui en quelque sorte se contient lui-même.

Dans la présentation qui suit, nous avons cependant épuré la présentation des spirales[18] en n'en retenant dans un ordre logique que celles qui ont concouru directement au résultat final.

5.3.1 Dictionnaire base

Dans cette première (figure 5.11) spirale on considère seulement la prise en charge des concepts, propriétés et relations.

Pour modéliser les relations il y a plusieurs alternatives. La définition en LDD (langage de description de données) de SQL considère la possibilité de définir une contrainte pour spécifier la référence. Dans ce cas, la relation garde le référence aux concepts ainsi qu'aux propriétés qui la définissent.

Un autre possibilité est celle de l'ORM pour qui la relation est un type spécial de propriété qui pointe vers un autre concept (notion d'identité). Dans ce cas, celui que nous avons retenu dans notre contexte de modélisation objet-relationnel nous avons une relation d'héritage (spécialisation) de « Propriété » dans une référence au « Concept » .

Le deuxième spirale (figure 5.12) considère l'existence des modèles, comme chaque concept appartient à un modèle, nous avons utilisé une relation de composition. « Model » est un concept car nous pouvons avoir avoir plusieurs modèles dans notre dictionnaire. Rappelons que du point de vue de Datarun, un modèle correspond à une application du SI.

Quand il y a trop des concepts dans un modèle, le modèle devient illisible. Un domaine est un collection de modèles reliées par une thématique particulière. Pourtant le mot « Domaine » est utilisé de multiples façon dans la littérature, et on voulait avoir une signification spécifique. Donc, nous avons choisi « Projet » simplement pour clarté linguistique. Parler de domaine ou de projet est équivalent (figure 5.13) .

Il faut remarquer que de toute façon le système peut accueillir le diagramme MCCD (il est considéré comme un diagramme standard). Mais on préconise la création d'un référentiel

18. Un peu comme l'exprime l'image québécoise : nous nous sommes arrangé avec le gars des vues (autrement dit nous avons réarrangé la réalité un peu comme au cinéma (vue = film).

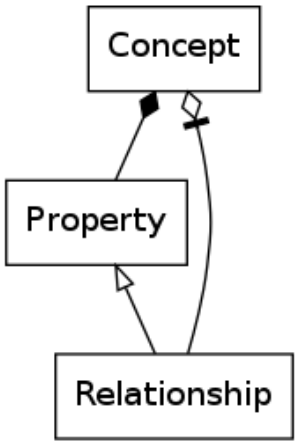


FIGURE 5.11: Dictionnaire de base. Spirale 1

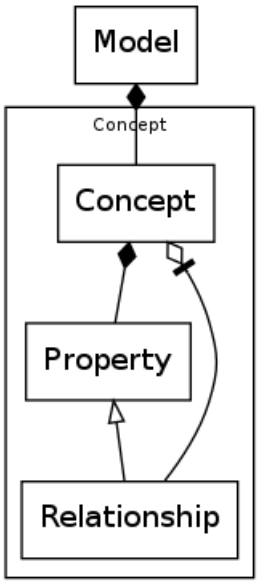


FIGURE 5.12: Dictionnaire de base. Spirale 2

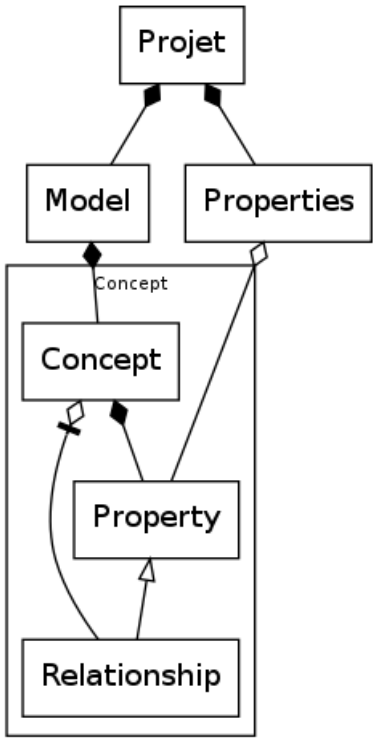


FIGURE 5.13: Dictionnaire de base. Spirale 3

commun composé par toutes les propriétés présentes dans le domaine, ce qui correspond à la création d'un dictionnaire pour le langage omniprésent mentionné par Evans.

Avec le concept de « Projet » apparaît le besoin d'avoir le tout comme un référence complète. L'approche Datarun préconise la définition d'un supraModel MCCD (modèle corporatif conceptuel des données). DDD mentionne la nécessité d'un langage omniprésent. Nous considérons qu'un dictionnaire général de toutes les propriétés est un bonne façon d'avoir une vision globale du projet.

La liste propriétés et ses équivalents forment la liste organisée de termes utilisés dans le projet et représentent le **thésaurus** du langage de domaine (figure 5.14) .

Par analogie avec UML, un package (ou paquetage en français) est un groupe de concepts, dans le but de les grouper à l'intérieur d'ensembles cohérents. Un package peut contenir a son tour d'autres packages. Le package a deux finalités principales (figure 5.15) :

- Facilitée graphique : un package permet de voir une hiérarchie de concept comme un seule concept. Donc, il peut mapper les relations des concepts qu'il contient comme s'il s'agissait d'un seul concept. Ils permettent de structurer les diagrammes et donnent une vision globale plus claire.
- Facilitée de développement : Un package peut avoir une « Interface » associée pour permettre l'évolution en parallèle de projets très vastes. Chaque groupe peut travailler

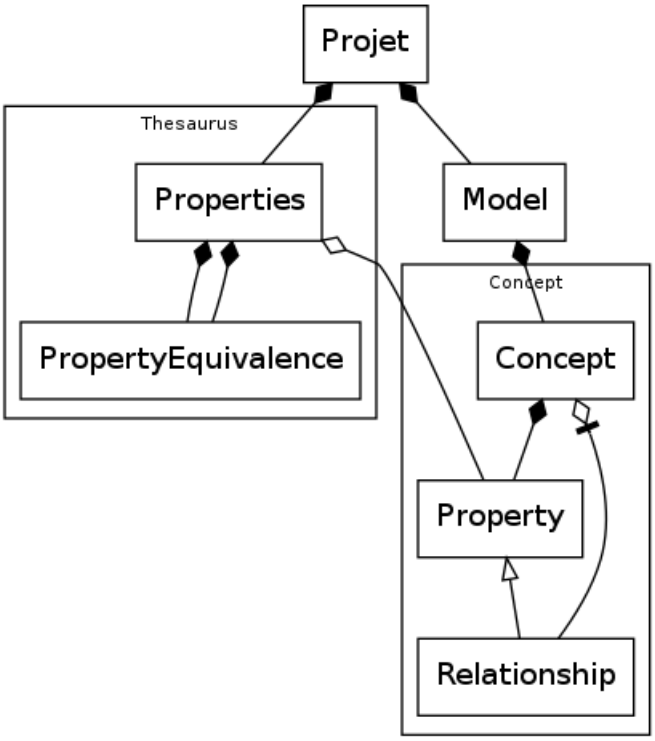


FIGURE 5.14: Dictionnaire de base. Spirale 4

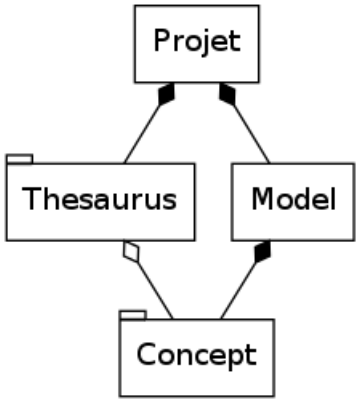


FIGURE 5.15: Dictionnaire de base. Spirale 5

et modifier les modèles a l'intérieur des packages à condition que cela n'affecte pas l'interface.

On voit comme le modèle devient plus simple en cachant les détails qui ne sont pas important pour communiquer l'idée centrale que l'on veut transmettre [19].

19. Note : Les packages n'ont pas était complètement implémenté dans la version actuel du prototypeur. Donc, on n'ajoute pas les modèles correspondent.

5.3.2 Hiérarchie, récursivité et autres idées

À un moment donné pendant le projet nous avons considéré l'option d'une hiérarchie des modèles. Après plusieurs itérations et de longues discussions, nous avons abandonné cette option en considérant que c'est une abstraction très difficile à trouver dans le réel perçu. Ainsi de nombreux modèles ont était explorés puis abandonnés pendant les processus de raffinement de notre structure d'accueil.

Dans la modèle actuel il existe un série objets intermédiaires pour gérer d'autres aspects tel que la sécurité et la création dynamique des vues. Mais ce sont surtout des aspects techniques d'implémentation et le prototypeur c'est aussi soumis à un cycle permanent de révision et modification. Nous ne détaillerons pas ici ces aspects.

5.3.3 Définition de classes à partir du modèle

Avec cette démarche de modélisation nous avons réalisé une application directe de la structure du modèle sur des classes dans un ORM. Nous avons choisi l'ORM Django. Voici un exemple de la définition de modèles dans le framework Django/Python.

```
from django.db import models

class MetaObj(models.Model):
    objType = models.CharField(max_length=50)
    code = models.CharField(blank = False, null = False, max_length=200)
    category = models.CharField(max_length=50, blank = True, null = True)
    alias = models.CharField(blank = True, null = True, max_length=50)
    physicalName = models.CharField(blank = True, null = True, max_length=200)
    description = models.TextField(blank = True, null = True)
...

class Domain(MetaObj):
    origin = models.CharField(blank = True, null = True, max_length=50)
    superDomain = models.ForeignKey('Domain', blank = True, null = True)

...
class Model(MetaObj):
    modelPrefix = models.CharField(blank = True, null = True, max_length=50)
    domain = models.ForeignKey('Domain')
```

5.3.4 Document de spécification du modèle

La structure d'un document de spécification est un arbre, une structure hiérarchique qui peut être sérialisée dans n'importe quel format. Comme exemple nous présentons la même structure

en XML qu'on avait défini (4.2) ainsi qu'une représentation graphique de l'arbre généré avec un outil de manipulation XML (5.16) .

Il existe un standard publié por l'OMG sur le format pour l'échange de modèles. Le XMI (acronyme de « XML Metadata Interchange"). L'usage le plus commun de XMI est l'échange de modèles UML, bien qu'il puisse être aussi utilisé pour la sérialisation de modèles d'autres langages (métamodèles). XMI est sûrement le chemin à suivre, mais pour le moment l'implémentation de ce standard est hors la portée de notre projet, et nous utilisons une structure XML/Json plus adapté à notre projet.

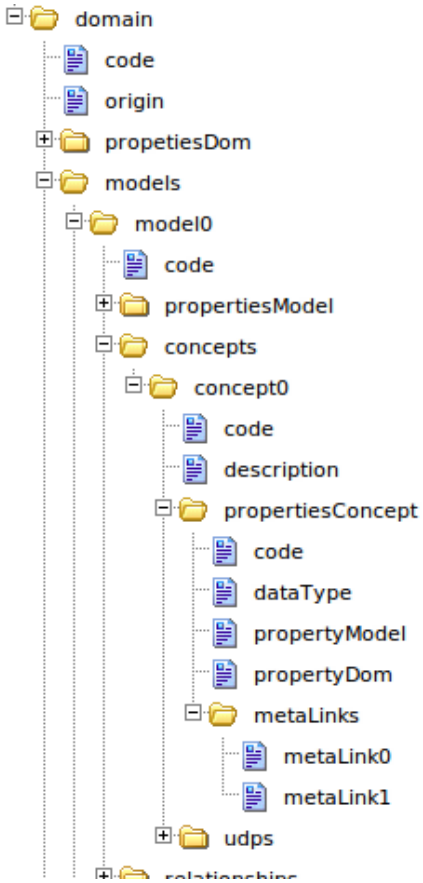


FIGURE 5.16: Représentation graphique d'un document de spécification

5.4 Les Vues

La spécification d'un modèle de définition d'interface est un étape importante de notre projet. L'objectif est de définir une interface utilisateur avec un structure riche dérivée des concepts du MCD. La méthode Datarun définit cette étape comme le « Modèle de spécification d'interface : MSI ». Dans notre projet nous utilisons la « vue » pour décrire l'interface. Les vues sont des éléments dynamiques d'interaction entre l'utilisateur et les constituants de notre prototype.

Tandis que le « MSI » est la représentation statique de la vue sous la forme d'un modèle de donnée. Une vue est en même temps la forme que l'utilisateur voit, mais aussi le code qui l'opérationnalise.

Un MSI décrit la composition d'un module du point de vue de l'utilisateur, ceci implique que ses composantes sont établies en fonction du comportement souhaité en respectant cependant la structure du modèle conceptuel de données MCD. Ainsi nous utilisons ici MSI et vue comme synonymes.

Le principes de base d'un MSI sont les suivants :

1. Un MSI peut faire un point d'entre (option de menu) dans le système.
2. Un MSI a un concept de base.
3. Un MSI peut contenir des propriétés appartenant au concept de base, ou propriétés dérivées des concepts inclus (références par clé étrangères). Étant donné que le concept référence est inclus, toutes les propriétés son visibles et utilisables mas non éditables.

    a) Une propriété est considérée dérivée par référence quand elle appartient à un concept inclus et la propriété n'a pas un support physique dans le concept de base. Cet mécanisme est appelle aussi absorption.

    b) Une propriété est considérée dérivé par copie quand elle existe dans le concept de base et elle reçoit la valeur d'une propriétés qui appartient à un concept inclus au moment de la création de l'enregistrement.

4. Un MSI permet d'éditer seulement le concept de base, les propriétés que n'appartient pas au concept de base ne sont pas modifiables.
5. Un MSI permet de naviguer sous la forme de Master-Detail pour un ou plusieurs de ses concepts dépendants. La définition de la navigation est limitée à la relation hiérarchique définie par la relation des entités. La navigation se produit par moyen de l'enchaînement des MSI.

La figure 5.17 représente le modèle retenu pour le MSI .

- La vue dépend d'un concept (concept base)
- La vue a des enfants
- La vue a des propriétés
- La vue serve de lookup aux autres vues
- Les propriétés des vues (VueProperty) sont dérivées des propriétés du concept base.
- Il existe un spécialisation la propriété de vues (ViewPropertyRef) qui pointent vers d'autres vues (références et propriétés dérivées).

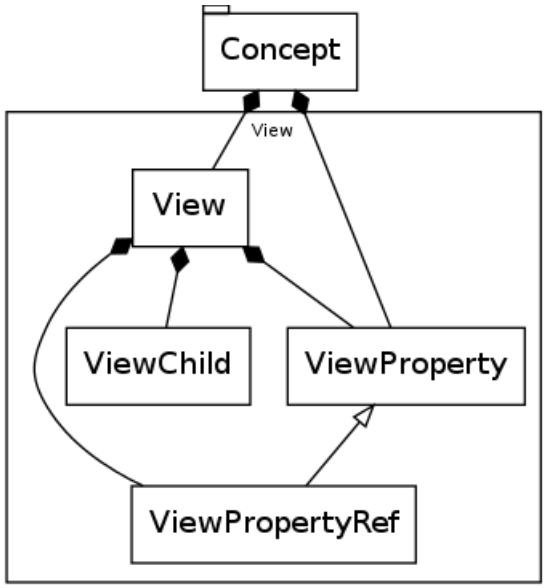


FIGURE 5.17: Modèle du MSI

5.4.1 Conceptualisation hiérarchique

De la même façon que nous avons défini un conceptualisation hiérarchique pour les MCD qui était matérialisée dans les documents de spécification du modèle on fait un définition hiérarchique de la structure de la vue (5.18) que sert de base pour la communication et définition de l'environnement client.

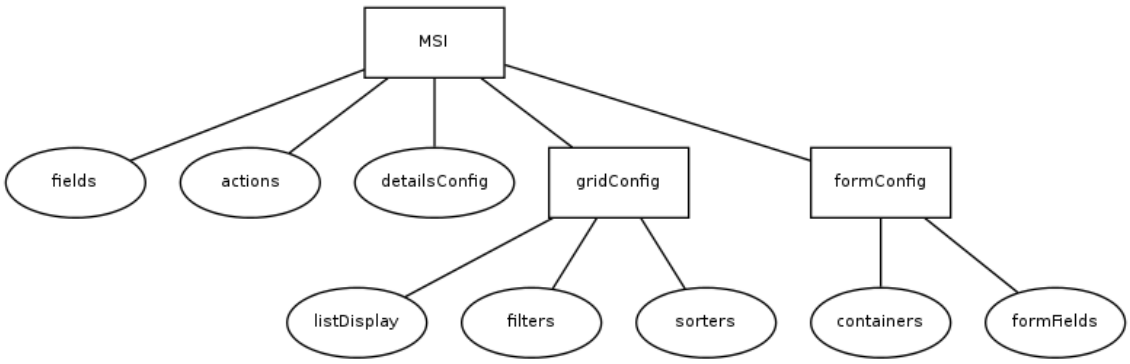


FIGURE 5.18: Structure du MSI

La structure du MSI est composé de :

- Configuration des propriétés (fields)
- Configuration des détails pour la navigation parent - enfants (detailsConfig)
- Configuration de la présentation en liste. (gridConfig)
- Configuration de la forme (formConfig)

– Configuration des actions associées à la vue (actions - ceci était hors de la porté de notre projet. Mais on l'utilise pour la configuration du prototypeur « bootstrapping »)

5.4.2 Construction

Pour la construction des MSI nous avons établi les étapes suivantes :

1. Dérivé d'un concept et assigner un nom.
2. Rajouter des éléments
   - Rassembler des colonnes provenant de plusieurs concepts parent (dérivés par référence et dérivés par copie).
   - Lier les vues enfants pour implémenter la navigation parent-enfant.

5.4.3 Proposition de représentation graphique

De la même façon qu'on doit représenter un modèle, une vue peut aussi être représenté graphiquement. Nous proposons de le faire en utilisant les notations définies précédemment (figure 5.19), il est toutefois possible de choisir une autre représentation comme celle qui est proposée dans Datarun.

Identifiant de la vue

Le identifiant de la vue est toujours composé par « nom du concept de base » + « . » + « nom de la vue ». Par exemple, si le concept de base est « Facture » et le nom de la vue est « fac ». Donc, l'identifiant de la vue est « Facture.fac ». Dans un diagramme, il est très facile de reconnaître un vue d'un concept. La différence la plus significative est que l'identifiant de la vue est toujours composé de deux parties [Concept.viewName].

Éléments graphiques

Les éléments graphiques (figure 5.19) pour représenter un MSI sont :

- La vue : Au centre du diagramme on trouve la représentation de la vue (son identifiant et ses propriétés).
- Le concept de base : Dans la partie supérieur du diagramme on trouve le concept de base. La relation avec le concept de base est une relation de composition.
- Les zooms : Les vues qui servent comme des références (souvent référé dans la littérature comme lookup ou zoom). La relation avec les parents est une référence entre vues, les concepts de composition et agrégation ne s'appliquent pas. Ainsi, le symbole pour indiquer une référence vers un parent est un flèche ouverte.

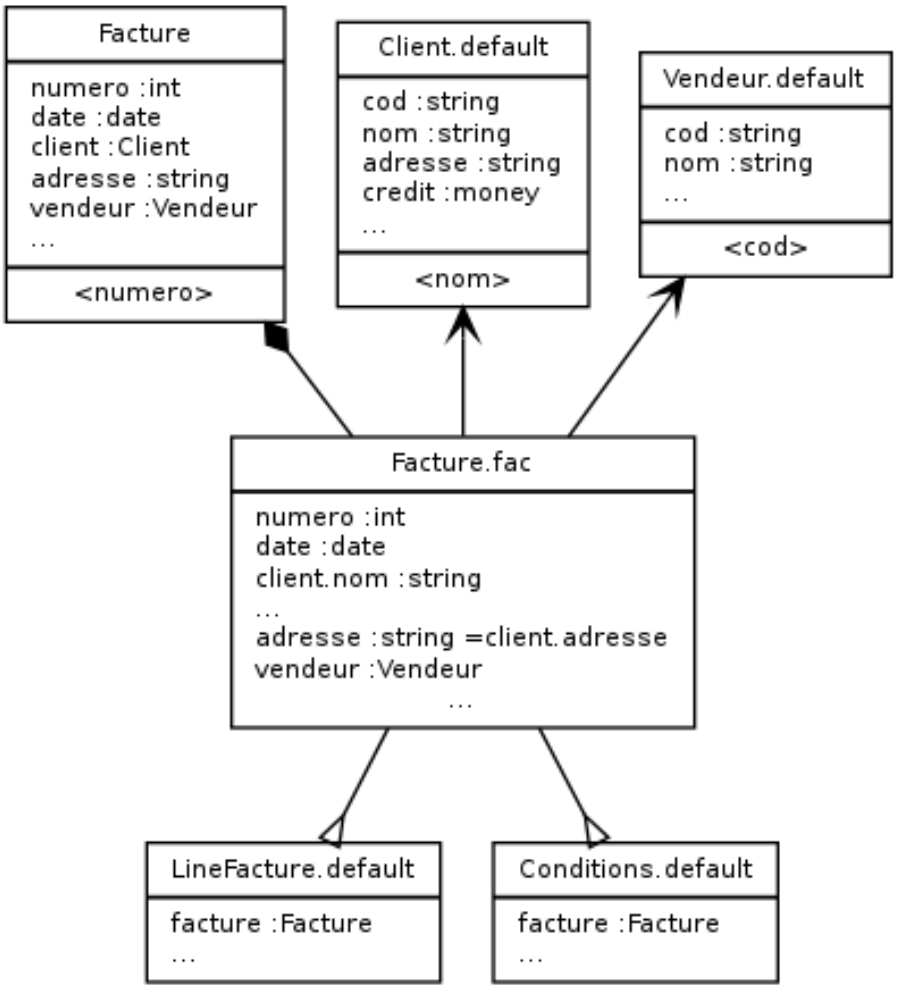


FIGURE 5.19: Schéma MSI

– Les Enfants : Les vues enfants qu'on veut avoir comme détails de notre vue centrale. Un relation parent enfant implique un parent et normalement plusieurs enfants. Le symbole pour indiquer une relation vers des enfants est un triangle vide avec la base du côté enfant (il se ouvre vers les enfants) qui rappelle la notation de pied de poule.

Convention pour la formation des noms des propriétés dans la vue

Pour les propriétés appartenant au concept de base, le nom des propriétés est égal a celui de son concept de base. Par exemple :

– Facture.fac

  – numero : int

  – date : date

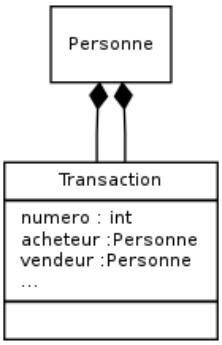


FIGURE 5.20: Plusieurs relations au même concept

Pour les nom des propriétés dérivées par référence le nom est compose du nom du concept de base plus un point plus le nom de la propriété du concept de référence. Par exemple :

- Facture.fac
  - ...
  - client.nom : string

Dans ce cas on dérive par référence le nom du client. Il faut faire attention on n'utilise pas le concept, on utilise le nom de la propriété que fait référence au concept. Parce que dans un concept on peut avoir plusieurs références au même concept. Prenons le cas d'une transaction de vente qui se fait entre deux personnes. Si on doit dériver des propriétés de "Personne" il faut savoir à quelle instance on fait référence (figure 5.20) .

Pour les propriétés dérivées par copie, il existe un propriété de base, donc elle sera affiche avec un symbole qui indique la propriété de référence d'où tirer son valeur initiale. Par exemple :

- Facture.fac
  - ...
  - adresse : string = client.adresse

La dérivation par copie fonctionne comme une valeur initiale au moment de la création de l'enregistrement. Si dans le cas de la facture, le client change son adresse, la facture garde l'adresse originale qui a été utilisée lors de la création. La dérivée par référence ne garde pas la valeur et si le client change son adresse, la facture présente la nouvelle adresse, ce qui est une erreur.

Pour la propriété de référence, elle utilise pour l'affichage ce qui était défini comme la "clé sémantique" dans le concept. Par exemple :

- Facture.fac
  - ...
  - client : Client

- vendeur : Vendeur

C'est l'équivalent de :

- Facture.fac
  - ...
  - client.nom
  - vendeur.cod

Car dans cet exemple, la "clé sémantique" de Client est le nom, et la "clé sémantique" du Vendeur est le code.

Références aux parents "Zoom"

La référence plus simple à un parent peut être une liste déroulant (comboBox) avec une paire de valeurs, la clé pour garder la référence et le nom pour indiquer quelle est la valeur. Nous croyons que ce n'est pas assez, par exemple, si la liste de valeurs est trop longue (plus de 20 enregistrements peut être déjà considérée comme longue) n'est pas convivial pour l'utilisateur d'effectuer une sélection. Il est possible que la liste de sélection soit un concept intermédiaire et les valeurs qui peuvent être utiles aux utilisateurs ont besoin d'être absorbées par d'autres concepts.

En résumé, nous pensons que la liste de valeurs pour faire un sélection devrait être aussi à une vue. Nous appellerons "Zoom" l'action d'aller cherché une référence, car cette une action qui permet de se "rapprocher" vers la vue de référence (toujours un parent dans la relation).

Un zoom permet d'entrer dans un vue différente pour effectuer quelques opérations de base sur elle. Par exemple, l'utilisateur peut effectuer une recherche en combinant différents critères, puis trier et finalement s'il n'a pas trouvé ce que il cherche, il pourrait créer l'enregistrement de référence dont il a besoin. Tout ceci, à l'intérieur de l'action "zoom".

5.4.4 Format de présentation

La structure de la vue n'impose pas de restriction sur le format de présentation. À priori il y a deux possibilités pour présenter l'information :

Tableau (grille)

Un tableau, ou gille, ou liste, permet de présenter un groupe de colonnes comme un tableau. C'est la présentation par défaut (figure 5.21).

Dans l'exemple de la figure 5.21 on peut repérer le colonnes de la vue « sy-facturation » il présente des propriétés propres au concept « Facture » et propriétés qui sont dérivées depuis le concept « Client » notamment le nom du client.

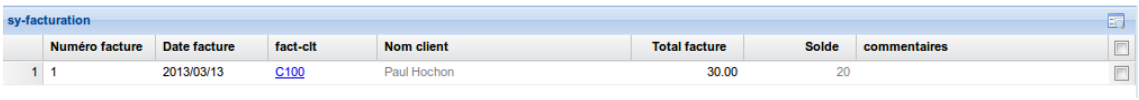


FIGURE 5.21: Présentation en tableau

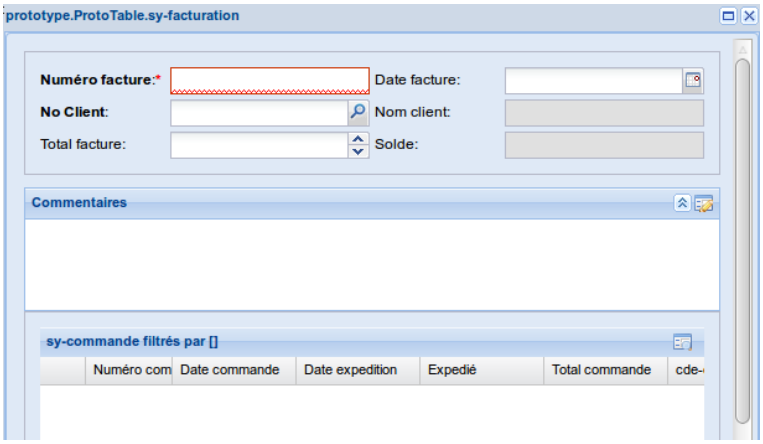


FIGURE 5.22: Présentation en Forme

Forme

La forme est utilisé pour toutes les opérations d'édition. Un forme correspond exactement à un enregistrement et il n'y a pas de différences conceptuelles dans le traitement de l'information. Un forme c'est simplement un réorganisation des propriétés à l'écran (figure 5.22). Dans cette exemple on peut aussi observer le détail de commandes par facture correspondant au concept « Commande » .

5.5 Définition des principales composantes des couches

Les principales composantes des couches BackEnd et FrontEnd sont présentées dans la figure .5.23..

- BackEnd
  - Fournisseur de services (à l'aide d'une URL)
    - Définir un format de donnée, pour envoyer et recevoir les messages (protocole).
    - Définir les paramètres d'appel ainsi que le méthode (GET-POST)
    - Définir le format de la réponse (XML-JSON)
  - Services
    - Menu de points d'entrée.

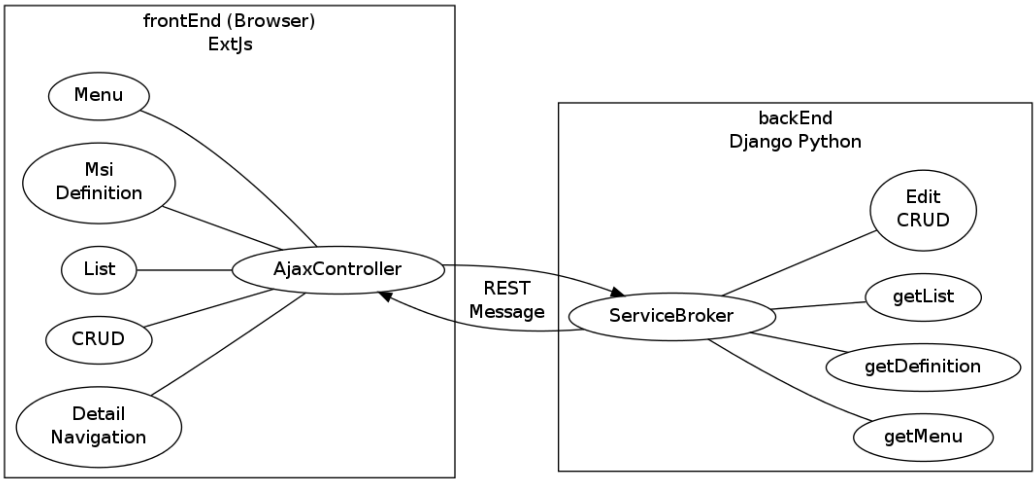


FIGURE 5.23: Objectifs spécifiques

  - Document de définition d'interface « MSI »
  - Requête de l'information correspondant un MSI spécifique (Liste de donnes) .
  - Edition (CRUD)
- Pour cela on utilise Django Python. Il est prévu que les modèles soient écrits en langage python directement sur le code de l'application Django. Le prototype utiliserait les modèles pour produire une application Web.

– FrontEnd

- Créer un module Web, avec une interface riche, ce qui permettra l'itération avec l'utilisateur grâce à l'interprétation des modèles d'interface MSI, et de présenter entre autres les fonctionnalités suivantes :
  - Créer un menu avec une visualisation hiérarchique des différents points d'entrée (modèles d'interface).
  - Pour chaque option sélectionnée, le système doit créer un élément graphique qui permet à l'utilisateur de visualiser les attributs du MSI sélectionné en forme de tableau. Selon le MSI, chaque option a la possibilité de définir des champs propres ou dérivés (Concepts liées à partir des clés étrangères).
  - Chaque colonne doit avec les entêtes définis par le MSI. Et l'ordre de colonnes doit suivre l'ordre défini par le MSI. Il peut avoir des attributs cachés que ne sont pas visibles par l'utilisateur.
  - Filtre de base : La liste initiale peut être filtrée pour des critères définis par le MSI, cette filtre est interne et ne peut pas être modifié par l'utilisateur. Par exemple la liste ne doit afficher que les enregistrements avec état = « Actif ».

- Filtre Initial : La liste initiale peut être filtrée par des critères définis par le MSI, cet filtre a une valeur par défaut et peut être modifié par l'utilisateur. Le filtre initial ainsi que les filtres que l'utilisateur fait sur l'écran sont toujours affectés par le filtre de base. Par exemple une liste de clients peut commencer par les clients dont le nom de famille commence pour « A ».
- Classement initial : La liste initiale peut être classée selon un ordre prédéfini par le MSI. (alphabétique, chronologique, etc..)
- Navigation : Chaque option doit être capable de naviguer sous la forme de Master-Detail pour un ou plusieurs de ses concepts dépendants. La définition est limitée à la relation hiérarchique définie par la relation des concepts.
- Une option détail, peut être elle-même promue comme option principale, en conservant le filtre défini par son ancien parent. De cette façon on peut être en mesure de naviguer à travers tous les niveaux de la hiérarchie.
- Vues alternatives : Chaque concept peut définir un ou plusieurs groupes de colonnes disposées dans un ordre spécifique. Ce qui est équivalent a cacher ou montrer des propriétés et les organiser de façon déterminée. L'utilisateur aura la possibilité de changer l'affichage pour voir uniquement les champs souhaités.
- Fiche : Chaque "MSI" aura une forme associée. La sélection d'un enregistrement dans la liste de propriétés permet d'afficher la forme associée.
- La forme associée permet l'édition (CRUD) pour la manipulation de l'information.

5.6 Services et Messages

La préparation d'une application web commence pour la séparation de responsabilités entre le client et le serveur. Dans un site traditionnel basée sur HTML c'est le serveur qui s'occupe de fournir de pages plutôt statiques. Depuis l'arrivé d'Ajax[133] [81] l'utilisation de langages de script permettent une interaction plus fluide avec le serveur. D'autres dispositifs comme de tablettes et téléphones mobiles partagent la même flexibilité qui est à la base de la Web 2.0.

Les avantages d'un style d'architecture basée sur des couches sont multiples[11] [72] [92]. La spécialisation de couches de notre système nous permet de faire évoluer l'outil de FrontEnd indépendamment de l'outil de BackEnd, en fait on utilise des technologies totalement différentes qui partagent un mécanisme de communication et une définition de messages commun.

Un service de façon générique, est un mécanisme permettant la communication et l'échange de données entre deux systèmes par moyen d'un protocole commun [20]. Les services sont au-

20. Il existe plusieurs technologies derrière les services web, mais il y a principalement deux differents facons de les implementer soit SOAP (Simple Object Access Protocol) ou REST (Representational State Transfer). Pour notre besoin, on a choisi REST pour un simple raison, c'est le plus simple.

tonomes, c'est à dire chaque service est maintenu, développé et déployé de façon indépendante. Comme chaque service est indépendant des autres services il peut être remplacé ou mis à jour sans affecter les applications qui l'utilisent. Voici des exemples de services utilisés par l'application :

5.6.1 Menu

A la demande **getMenu** du client, le serveur répond avec le message suivant :

```
[{
    "text": "Dictionnair\u00e9 des donn&eacute;es",
    "expanded": true,
    "children": [{
        "text": "Model",
        "iconCls": "icon-model",
        "leaf": true,
        "id": "protoExt.Model",
        "index": 1
    }, {
        "text": "Elements De Donnees",
        "iconCls": "icon-property",
        "leaf": true,
        "id": "protoExt.Property",
        "index": 2
    }, {
        "text": "Entite",
        "iconCls": "icon-concept",
        "leaf": true,
        "id": "protoExt.Concept",
        "index": 3
    }],
    "index": 1
}]
```

PCI - Proto Concept interface

A la demande **getViewDefinition** du client, le serveur répond avec le message suivant :

```
{
    'metaData': {
        'storeFields': 'code,description',
        'description': 'Model',
        'shortTitle': 'Model',
        'protoSheet': {},
        'protoIcon': 'icon-model',
        'fields': [{
            'header': 'Vues',
```

```
            'type': 'CharField',
            'name': 'code',
            'width': 300
        }, {
            'flex': 1,
            'type': 'TextField',
            'name': 'description',
            'header': u 'description'
        }],
        'initialFilter': {},
        'initialSort': [],
        'protoViews': [],
        'protoFilters': [],
        'protoDetails': [{
            'menuText': 'Élements de données',
            'conceptDetail': 'protoExt.property',
            'detailField': 'concept__model__pk',
            'masterField': 'pk'
        }],
        'sortFields': ('code', 'description'),
        'searchFields': ('code', 'description')
        'successProperty': 'success',
        'totalProperty': 'totalCount',
        'idProperty': u 'id',
        'root': 'rows',
    }
}
```

5.7 Le code du prototypeur

Le code du prototypeur est publié sous licence GPLv2 et suivantes et il est disponible dans lien suivant : `https://github.com/certae/ProtoExt`

Dans l'annexe A, nous présentons un résumé qui donne une idée du travail.

Chapitre 6

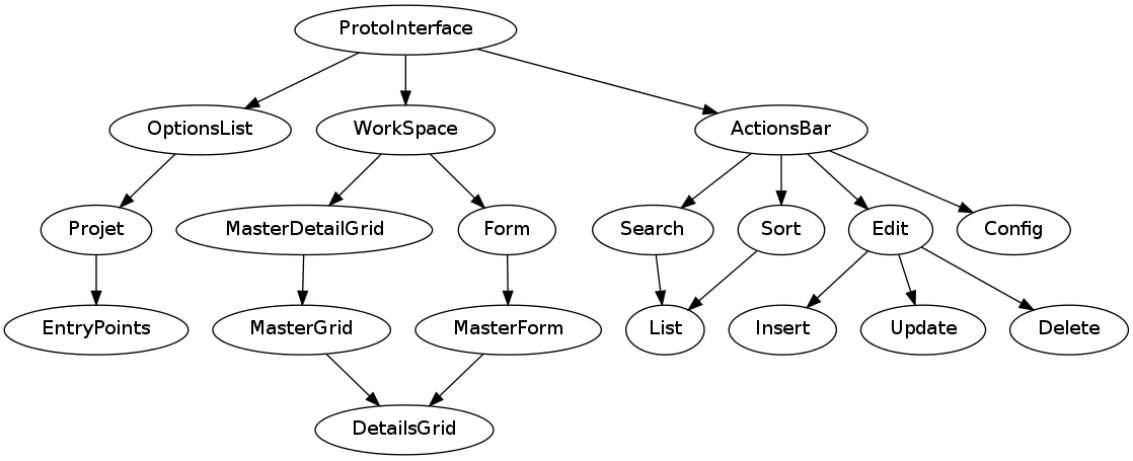
L'interface de l'application Prototypeur

6.1 Interface utilisateur

La conception fonctionnelle détaillée d'un système doit contenir au moins le modèle conceptuel de domaine ainsi que la définition précise des interfaces (essentiellement les interfaces interactives de saisie et de consultation des données). La définition précise des interfaces peut avantageusement être concrétisée par le moyen d'un prototype.

Une interface, qui assure l'interaction avec les utilisateurs, est une des principales composantes d'un système. La structure de notre interface est décrite dans le diagramme conceptuel de la figure 6.1. Qui de manière générale indique :

- L'interface (ProtoInterface) est divise en trois sections
  - OptionList : corresponds au menu d'options d'utilisateur. Ici, on doit trouver la liste des points d'entrée au système divisé par projet.
  - WorkSpace : correspond à l'espace de travail, il doit permettre d'ouvrir une ou plusieurs options de style parent-enfant organisées à l'aide des onglets. Il est possible d'afficher les formes de dialogue avec l'utilisateur pour les actions d'édition. Les formes et les listes parents auront aussi la possibilité d'avoir des listes enfants.
  - ActionBar : La barre d'actions contient les possibles actions permises à l'utilisateur pour un option déterminée. La barre d'état est synchronisée avec l'espace de travail, nettement avec les listes parent-enfant. Les options d'édition sont affichées dans le bord supérieur de chaque liste.



Q

FIGURE 6.1: Schéma de l'interface

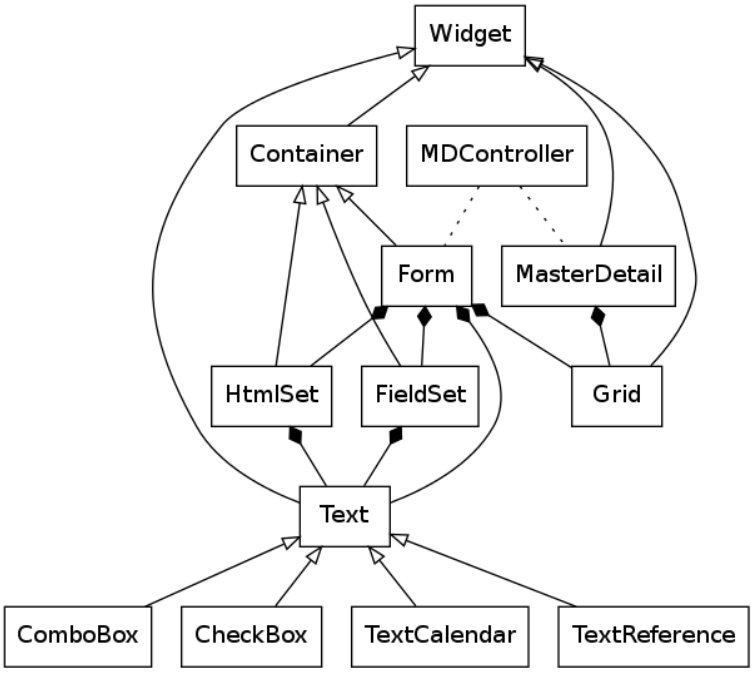


FIGURE 6.2: Widgets

6.2 Widgets

Une des parties les plus importantes dans le cadre d'exécution correspond à la définition des éléments de représentation des données pour l'interaction avec l'utilisateur. Sencha ExtJs offre un extraordinaire éventail de widgets [1] pour configurer l'interface utilisateur. Et il est possible la spécialisation des composants à partir de la bibliothèque de base pour les adapter à nos besoins. Les widgets sont automatiquement [2] sélectionnés par le prototypeur en fonction des types de base et sont utilisés pour manipuler l'information d'entrée et sortie.

Dans le processus de développement de l'application, nous avons défini une série d'objets de haut niveau qui peuvent être utilisés pour caractériser les interactions de l'utilisateur au sein de l'interface utilisateur. Voici les éléments que nous avons dû adapter pour l'espace de travail (figure 6.2) :

- **MasterDetail :** C'est l'élément par défaut pour la présentation des données. Il est composé d'un liste « parent » et optionnellement d'un ou plusieurs listes « enfants » qui sont filtrés selon l'élément sélectionné dans la liste « parent » [Zone 1 et 2] de la figure 6.3.
- **Container**. C'est un conteneur d'autres éléments de présentation. Les conteneurs sont généralement utilisés pour agir comme des éléments de regroupement avec des contraintes de mise en page associés aux objets d'interaction connexes. Les conteneurs qu'on utilise sont :

  1. Form : Utilisé pour les formes d'édition. Une forme peut regrouper des « fieldsets », « htmlsets », « grids » et des widgets spécialises à partir de « text ». Les « grids » à l'intérieur d'une forme sont seulement des enfants (détails) du registre principal (main record) .
  2. FieldSet : C'est un conteneur générique utilisé dans les formes pour regrouper principalement des « text ». Mais il peut contenir « grids » et « htmlsets ». Le « fieldset » serve pour définir la disposition des éléments dans la forme.
  3. HtmlSet : C'est un conteneur spécifique pour l'édition WYSIWYG [3] qui est par la suite enregistré en format HTML. Il peut contenir que des types « text » de base qu'il transforme en un éditeur HTML.

- **Text**. Le widget de texte correspond directement à un propriété d'un concept particulier du modèle. Il est utilisé pour afficher et saisir des données. Un exemple typique est un champ de texte dans laquelle l'utilisateur peut saisir des données alphanumériques comme les noms ou les salaires. Il y a des spécialisations selon le type de base :

1. Widgets : Windows gadgets
2. Il est possible de changer le comportement prédéfini au moment de la configuration des propriétés.
3. WYSIWYG acronyme pour "What You See Is What You Get".

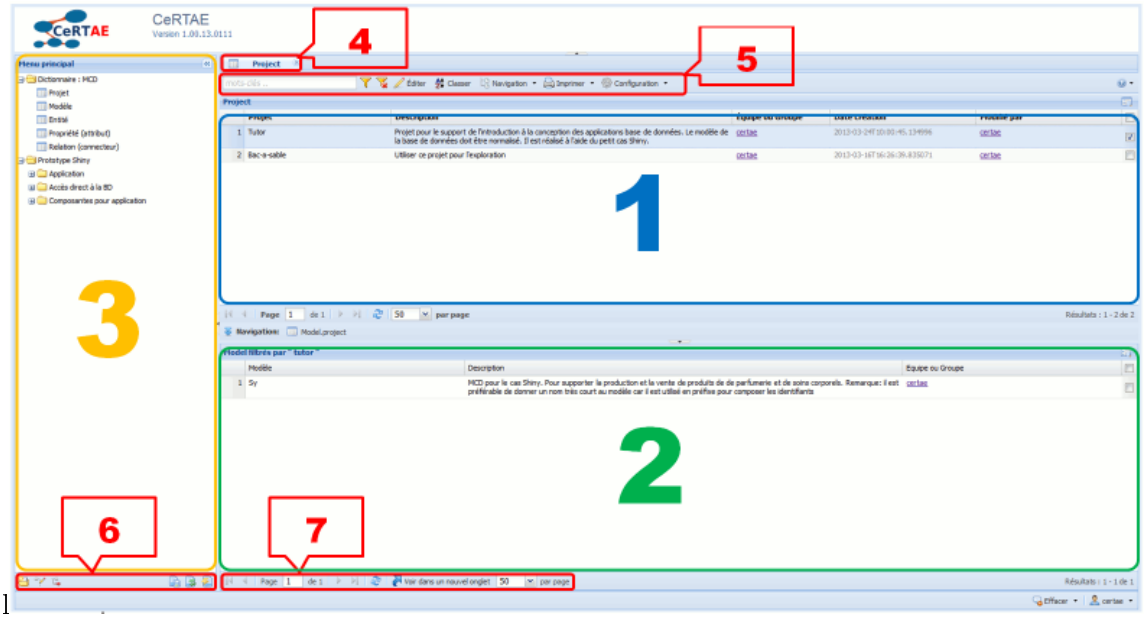


FIGURE 6.3: Organisation de l'interface de l'application

- ComboBox : Permets la sélection d'une liste prédéfinie dans le modèle. (i.e. État civil : Célibataire, marié, ...)
- CheckBox : utilisé pour les propriétés de type booléennes (oui/Nom) .
- TextCalendar : utilisé pour les propriétés de type date. Permet la sélection de la date à partir d'un calendrier.
- TextReference : utilisé pour sélectionner une référence à partir d'une relation. Permet ouvrir une liste avec les occurrences du concept parent (zoom). Par exemple, à partir d'un produit, sélectionner la famille.

– **Grid**. C'est une liste (des données tabulaires) d'occurrences de la vue sélectionnée. Permet les opérations de filtrage, triage, sélection entre autres.

6.3 Description de l'interface

La réalisation des composantes d'un tel diagramme à l'aide du logiciel retenu pour le FrontEnd entraîne différentes zones repérées dans la figure 6.3

6.3.1 Espace de travail

L'espace de travail [Zone 1 et 2] est au centre de l'interface, il sert à afficher le contenu des objets sélectionnés dans le menu latéral (OptionList ou EntryPoints) [Zone 3] où sont les points d'entrée pour le système. Quand on clique deux fois sur un objet du menu principal [Zone 3],

FIGURE 6.4: Boîte Bento

le contenu de cet objet est affiché sur la zone de travail [Zone 1]. Chaque fois qu'un objet est ouvert, un nouvel « onglet » apparaît dans la partie supérieure de la liste [Zone 1]. On peut se déplacer d'un contenu à l'autre en cliquant sur un onglet. La liste principale affiche le contenu de l'onglet sélectionné. Plusieurs onglets peuvent être ouverts en même temps. [4]

Ce nouveau style de design de l'espace de l'interface nous l'avons appelé « Boîte Bento » en allusion à la boite japonaise des repas rapides (figure 6.4).

L'espace de travail est constitué d'une ou plusieurs listes parent-enfant (masterDetail) qui sont séparées à l'aide d'onglets [Zone 4]. Chaque onglet contient une liste master-détail [Zone 1] qui contient à son tour une ou plusieurs listes enfants[Zone 2]. L'espace de travail contrôle aussi la barre d'actions[Zone 5] et la barre d'état [Zone 7].

La section des « listes enfants » [Zone 2] permet le filtrage automatique de la vue en fonction la ligne sélectionnée dans la liste parent [Zone 1].

On a établi la possibilité de naviguer d'un parent vers plusieurs enfants. Donc, la section « listes enfants » [Zone 21]) est décoré d'un barre de navigation (entre [Zone 1] et [Zone 2]) qui permet la sélection d'affichage de la « liste enfant » sélectionnée. La barre de navigation permet aussi d'afficher ou cacher la section des listes enfants.

6.3.2 Liste principale (MasterDetailGrid [Zone 1])

La liste principale (figure 6.5)est une vue qui correspond à la définition d'un MSI (le MSI comporte aussi la navigation vers les enfants). La barre de menu présente les options : recherche, éditer, classer, navigation, imprimer et configurer (les options disponibles dépendent des droits alloués aux utilisateurs). Situé à droite de la barre de menu, on trouve le bouton d'aide en ligne de l'application.

4. Il est déconseillé d'ouvrir trop (plus de dix à la fois), car cela peut avoir un effet sur la mémoire et ralentir l'application.

FIGURE 6.5: Liste principale

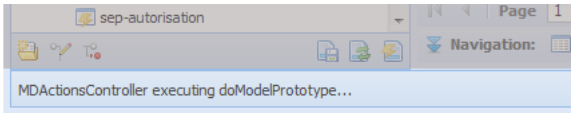


FIGURE 6.6: La barre des messages

Au-dessous du bouton d'aide en ligne de l'application se trouve le bouton pour ouvrir le formulaire de visualisation en mode de lecture. Pour se servir de cette fonctionnalité, on sélectionne une ligne de la liste et ensuite on clique sur le bouton. Une nouvelle fenêtre s'ouvre automatiquement. Les informations concernant l'objet sélectionné sont affichées dans la forme correspondante. D'autre part, la liste principale affiche le contenu de la vue sélectionnée dans le menu principal. Dans cet exemple [5], les informations affichées dans la liste correspondent aux informations de la vue « projet », donc la première colonne correspond au nom du projet, la deuxième la description du projet, la troisième c'est le nom de l'équipe ou du groupe propriétaire du projet, la quatrième c'est la date de création du projet et la cinquième correspond au nom de la dernière personne qui a fait des modifications dans le projet. La partie inférieure correspond à la barre de navigation des résultats.

6.3.3 Barre de messages

L'application du prototypeur communique avec l'utilisateur par l'entremise de la « barre de messages ». La « barre de messages » est située dans la partie inférieure de l'espace de travail. Quand l'application a besoin de communiquer avec l'utilisateur, pour la confirmation d'une opération ou un message d'erreur, elle le fait dans la barre de messages. Les messages d'erreur sont affichés en utilisant une police en couleur rouge.

Dans la figure 6.6, le message montre l'exécution de l'opération « doModelPrototype »et la figure 6.7 donne un exemple de message d'erreur. Par exemple un problème de communication avec le serveur.

5. Nous utilisons ici comme exemple la partie du prototypeur qui permet de saisir le modèle ; en effet, le prototypeur est construit avec lui-même.

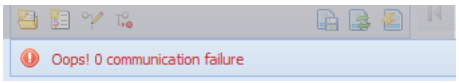


FIGURE 6.7: message d'erreur

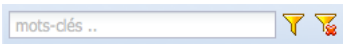


FIGURE 6.8: action filtrer

6.3.4 Barre de navigation des résultats [Zone 7]

Le « menu de navigation des résultats » regroupe les fonctions disponibles pour se déplacer entre les pages des résultats. Il est possible de choisir le nombre de résultats affichés par page, rafraîchir les résultats affichés sur la liste et pour les listes de détails ouvrir les résultats dans un nouvel onglet. Chaque liste a son propre menu de résultats, ils sont situés en bas de chacune.

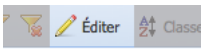
6.3.5 Barre d'actions (ActionsBar [Zone 5])

La barre d'actions [Zone 5] regroupe les fonctions disponibles pour la liste principale et la liste de navigation. Les fonctions sont les suivantes : filtre, édition, classement, impression et configuration. La fonction de configuration est disponible seulement pour les utilisateurs qui ont des droits de paramétrisation de l'application.

Filtrer

La fonction « filtrer » (figure 6.8) filtre le contenu de la liste principale de l'onglet sélectionné. Le filtrage s'applique sur plusieurs critères, par exemple, la colonne « nom » et la colonne « description » de l'onglet actif[6]. Les critères pour filtrer peuvent être personnalisés dans le menu de configuration, pour ce faire, il est nécessaire d'avoir un compte avec les permissions d'administrateur. Pour l'utiliser, on entre l'expression de recherche dans le champ texte du menu de fonctions. Les expressions peuvent être en majuscules ou minuscules. Attention : si le mot contient des accents, il faut respecter les accents, sinon, la fonction ne retournera pas des résultats.

- Cliquer sur le bouton « filtrer » du menu.
- Pour supprimer les critères de filtrage et réafficher le contenu complet sur la liste cliquer sur le bouton « supprimer filtre ».



...

FIGURE 6.9: action éditer

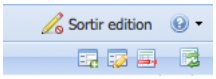


FIGURE 6.10: actions d'édition

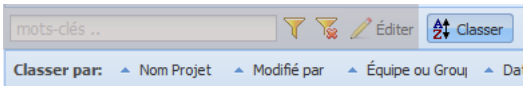


FIGURE 6.11: Action classer et son sous-menu

Éditer

L'action éditer (figure 6.9) permet de passer en mode édition, le mode édition change la configuration des options pour afficher les options qui permettent d'ajouter, modifier et effacer les enregistrements. Chaque liste dans une configuration père-enfant aura ses propres contrôles d'édition.

En cliquant sur le bouton éditer, on passe du mode consultation au mode d'édition et les contrôles d'édition apparaissent (figure 6.10) en haut à droit des deux listes. Quand le mode d'édition est actif, le menu des fonctions est caché et le bouton « sortir édition » est affiché.

Détails sur les contrôles d'édition :

-  Le bouton « Ajouter » sert à ajouter une nouvelle valeur (ligne) dans la liste sélectionnée. Dans la liste principale, la nouvelle valeur est ajoutée à la fin de la liste de valeurs. En cliquant sur le bouton « Ajouter », un formulaire apparaît à l'écran. Le remplir et cliquer sur « Enregistrer ».
-  Le bouton « Éditer » sert à modifier les valeurs d'une ligne dans la liste sélectionnée.
-  Le bouton « Supprimer » sert à supprimer les valeurs d'une ligne dans la liste sélectionnée.
-  Le bouton « Copier » sert à copier une ligne dans la liste sélectionnée.

Classer

La fonction « Classer » (figure 6.11), comme son nom l'indique, permet de classer le contenu de la « liste principale » de façon ascendante ou descendante par les différentes options de son

6. Nous utilisons ici le mot colonne en référence à la forme de l'interface, mais il s'agit bien d'une propriété du modèle objet-relationnel.

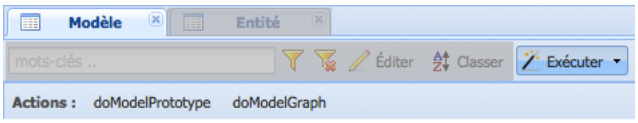


FIGURE 6.12: Action exécuter

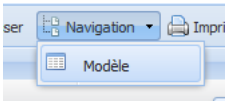


FIGURE 6.13: Action naviguer dans la barre d'actions

sous-menu. Les options du sous-menu « Classer par : » varient selon la définition de la vue (MSI).

Exécuter

La modélisation objet-relationnel permet d'assigner différentes méthodes à chaque vue. C'est la façon de modéliser les règles d'affaires. La création dynamique de règles d'affaires n'est pas dans la portée de notre projet, on a cependant la possibilité d'écrire les règles directement dans Django. Voici des exemples des règles et procédures que nous avons mis en place pour l'exécution du prototypeur (figure 6.12) :

- Vue du modèle : « doModelPrototype » génère les vues du prototype pour un modèle sélectionné. Cette démarche tient pour acquis que le modèle de données à prototyper est déjà complété : tous les concepts ont des propriétés et il y a des relations créées entre les concepts. Un message de confirmation de l'opération est affiché dans l'interface (voir la barre de messages pour plus de détail).
- Vue du modèle : « doModelGraph » génère le modèle conceptuel graphique pour un modèle sélectionné. À partir de l'onglet « Modèle », cliquer sur la ligne de la liste principale qui correspond au modèle à prototyper.

Naviguer

Cette action (figure 6.13) permet de naviguer vers les détails de la vue sélectionnée. Par exemple : à partir de l'onglet « projet » on peut naviguer vers les modèles disponibles pour ce projet. À partir de l'onglet « modèle », vous pouvez naviguer vers les concepts du modèle sélectionné. À partir de l'onglet « concept », la navigation se fait vers les propriétés, les relations, les vues pour l'entité sélectionnée.

Naviguer vers les détails d'un objet peut se réaliser de deux façons : par la fonction « Navigation » du menu des fonctions ou par le « menu de navigation » de la liste de navigation. Si la liste

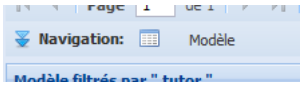


FIGURE 6.14: Barre de navigation

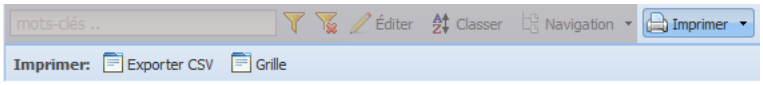


FIGURE 6.15: Action imprimer

de navigation est cachée, elle est affichée automatiquement à l'écran. La barre de navigation (figure 6.14) montre la liste de tous les modèles détails pour la vue sélectionnée.

- Pour voir les modèles d'un autre enregistrement, cliquer un autre enregistrement de la liste principale. Le contenu de la liste de navigation est actualisé automatiquement.
- Pour visualiser l'information d'un concept enfant de la liste de navigation, sélectionner le modèle sur la liste, ensuite, cliquer sur l'icône pour ouvrir le formulaire de visualisation ou d'édition suivant le mode activé.
- Pour faire disparaître la liste de navigation, cliquer sur le bouton situé en haut au milieu de cette même liste.

La « barre de navigation » se trouve dans la partie inférieure de la liste principale. Quand les deux listes sont affichées à l'écran, ce menu est situé entre les deux listes.

Imprimer

La fonction « Imprimer » (figure 6.15) permet d'imprimer le contenu de la liste principale. Le sous-menu « Imprimer » est composé de deux options : « Exporter CSV » et « liste ». On peut aussi accéder aux options du sous-menu en cliquant sur le petit triangle noir situé à droit du bouton « Imprimer ». Quand on clique sur l'option « liste », un nouvel onglet est ouvert dans le fureteur, la fenêtre du dialogue d'impression est affichée à l'écran. Selon le type d'imprimante installée, on peut enregistrer le résultat de l'impression en format PDF [7]. Le contenu de la liste est transformé dans un format de sortie en HTML (le résultat est montré dans le nouvel onglet ouvert). On peut aussi enregistrer le contenu de cet onglet comme une page HTML.

Quand on clique sur l'option « Exporter CSV [8] », une fenêtre est affichée à l'écran. À partir de cette fenêtre, il est possible d'ouvrir le fichier avec un logiciel de feuille de calcul [9] ou de l'enregistrer (figure 6.16).

7. Certains navigateurs offrent la possibilité d'enregistrer en format PDF sans avoir à installer un logiciel spécial
8. csv : comma-separated values
9. comme LibreOffice Calc ou MS-Excel

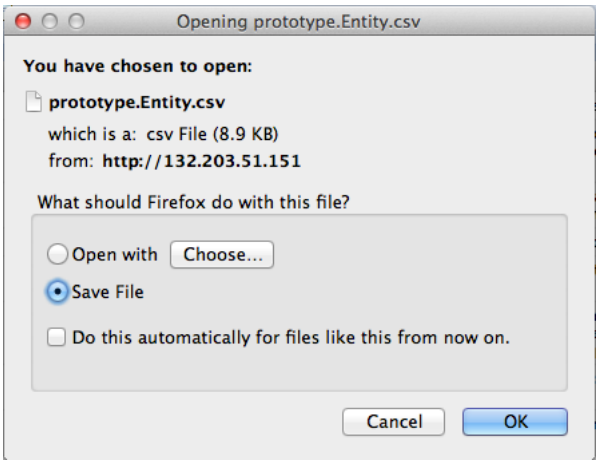


FIGURE 6.16: Exemple de la fenêtre exportation CSV

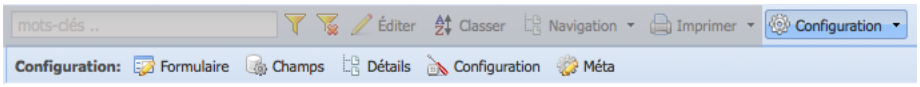


FIGURE 6.17: Barre de Configuration

Le contenu de la liste principale est imprimé ou exporté au complet. Par exemple, si on a 3 000 résultats qui sont distribués dans 10 pages, quand nous utilisons une des fonctions liste ou Exporter CSV, les 3 000 résultats seront imprimés ou exportés. Il n'est pas nécessaire de se déplacer à chacune de pages pour faire l'impression/exportation.

6.3.6 Barre de Configuration

La fonction « Configuration » (figure 6.17) permet l'accès aux fonctions de personnalisation de l'application du prototypeur. Quand on clique sur le bouton « Configuration », un sous-menu apparaît au-dessous du menu de fonctions. Ce sous-menu est composé des options suivantes : Formulaire, propriétés (Champs), Détails, Configuration et Méta. On peut aussi accéder aux options du sous-menu en cliquant sur le petit triangle noir situé à droit du bouton configuration.

L'accès à toutes les options de la fonction « Configuration » est réservé aux utilisateurs avec les droits de administration, si les droits ne sont pas suffisants, on voit uniquement l'option « Configuration » qui donne accès aux propriétés de configuration de base. Une présentation des options de configuration est donnée plus loin6.4.

6.3.7 Menu principal (OptionsMenu EntryPoints [Zone 3] [Zone 6])

Le « menu principal » (figure 6.18) situé du côté gauche de l'interface, affiche les points d'entré du prototype composé de la liste des projets et leurs vues prototypes. Pour afficher le contenu

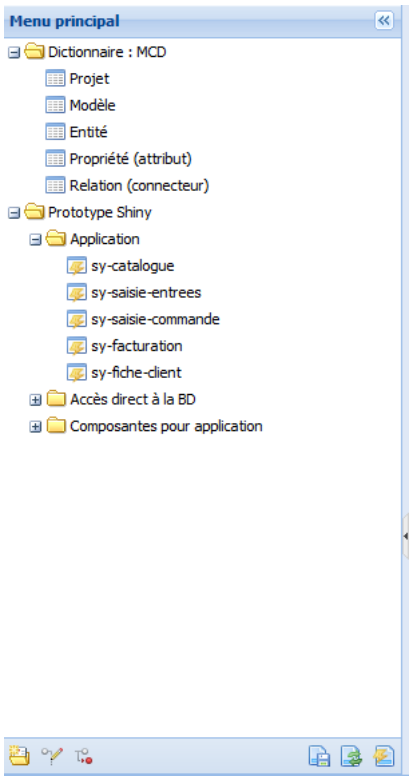


FIGURE 6.18: menu principal

d'un point d'entrée sur la liste principale, cliquer deux fois sur l'option à ouvrir. Un nouvel onglet est ouvert pour chaque option sélectionnée. Il est possible de cacher le menu principal en cliquant sur le bouton supérieur droit du menu ou sur la marge. Pour réafficher le menu principal, cliquer sur le bouton ou sur la marge.

Barre de configuration de menu [Zone 6]

Le « menu des fonctions du menu principal » regroupe les fonctions disponibles pour la configuration du menu principal. Les fonctions permettent d'y ajouter un nouveau répertoire, de modifier le nom d'un objet du menu, d'effacer un objet du menu, d'enregistrer la personnalisation du menu, recharger le menu et réinitialiser le menu. Voici la liste d'options disponibles :

-  Créer un nouveau dossier dans l'arborescence. Pour déplacer le nouveau dossier, cliquer dessus, sans relâcher le bouton de la souris, le glisser et ensuite lr déposer à l'endroit voulu.
-  Modifier le noeud permettre de changer le nom de l'objet choisi.
-  Sélectionner, supprimer noeud pour supprimer un objet dans l'arborescence.

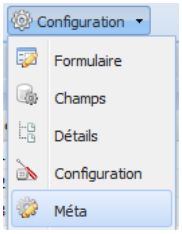


FIGURE 6.19: Configuration des métadonnées (menu)

-  Enregistrer menu sert à enregistrer l'état actuel du menu principal. Toutes les modifications (personnalisations) réalisées dans l'arborescence sont enregistrées, en cas contraire, un menu par défaut sera toujours affiché (et refait) par l'application.
-  Recharger menu fait un appel à l'application pour recharger la configuration du menu. Si les modifications effectuées ont été enregistrées, l'application retournera le dernier état enregistré du menu.
-  Réinitialiser menu fait un appel à l'application pour charger la configuration initiale du menu.

6.4 Configuration

Le prototypeur est un outil très flexible pour supporter la configuration des interfaces d'utilisateur. Il permet de définir la navigation parent-enfant, il permet aussi de faire les opérations d'absorption de propriétés provenant des concepts de zoom (look up, liste de sélection).

6.4.1 Option meta

L'option « Méta » (figure 6.19) donne accès à l'ensemble des propriétés de personnalisation de l'application prototypeur. Cette option est réservée aux utilisateurs avec des droits d'accès avancés (par exemple administrateur). À partir de cette fenêtre, il est possible de personnaliser la liste principale, les formulaires, les détails, les propriétés et d'autres objets de l'interface (figure 6.20).

Liste des propriétés de l'option « Méta »

Les options suivantes sont les plus utilisées lors de la personnalisation de l'interface. Les propriétés non mentionnées ne sont pas prises en compte dans ce guide d'utilisation.

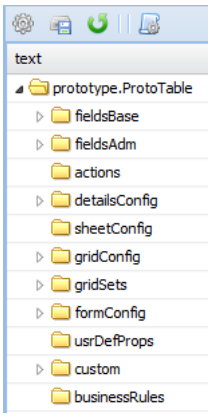


FIGURE 6.20: Configuration des métadonnées (arbre)

- fieldsBase : change la définition des propriétés. La liste des propriétés à modifier est assez complète, mais les propriétés le plus utilisées sont : fieldLabel, header, sortable, serchable, flex tooltip et wordWrap. (Voir : Comment personnaliser les propriétés 6.4.4).
- gridConfig : personnalise la présentation de la liste principale. La liste des propriétés à modifier est assez complète, mais les propriétés le plus utilisées sont : listDisplay, searchFields, sortFields et hiddenFields. (Voir : Comment personnaliser la liste 6.4.2).
- formConfig : personnalise la présentation du formulaire. Le formulaire peut être personnalisé à partir de deux endroits : formConfig dans l'option Méta ou dans l'option formulaire du menu de configuration. On recommande de se servir plutôt de l'option formulaire que de formConfig parce que l'option formulaire est plus conviviale pour la personnalisation. La liste des propriétés à modifier est assez complète, mais les propriétés le plus utilisées sont : tooltip, fieldLabel, labelAlign et hidden. (Voir : Comment personnaliser un formulaire 6.4.6)

6.4.2 Personnaliser la liste

Les exemples de personnalisation suivants montrent comment réaliser la configuration de base pour réussir une interface fonctionnelle et présentable.

Propriété « gridConfig »

- Ouvrer le contenu du dossier « gridConfig » (figure 6.21) en cliquant sur le triangle blanc situé à gauche du dossier.
- Sélectionner de la liste la valeur à personnaliser.
- Modifier les valeurs.

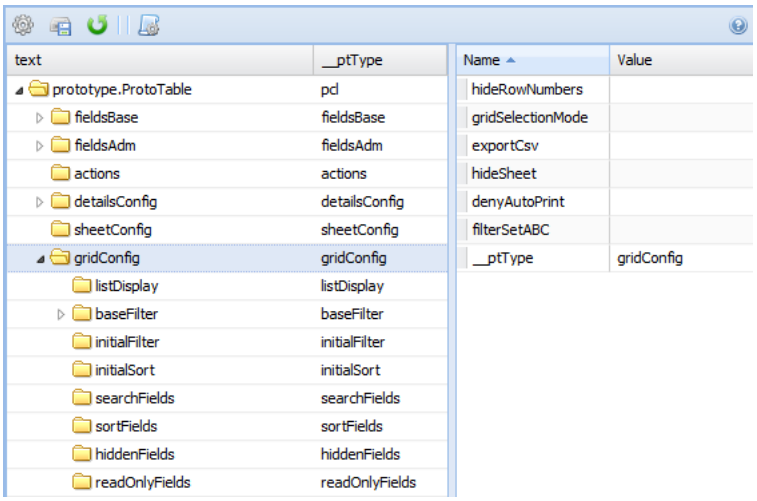


FIGURE 6.21: Personnalisation de la liste

– Ensuite, enregistrer les changements en cliquant sur le bouton « Mettre à jour la méta »

Les options plus communes dans la personnalisation de la liste (gridConfig) sont :

hideRowNumbers : valeur booléenne qui sert à afficher ou cacher sur la liste les numéros de lignes.

exportCsv : valeur booléenne qui sert à activer ou désactiver la fonctionnalité d'exportation en format CSV. Noter que la fonction exportCsv exporte la liste complète de résultats et non seulement ceux qui sont affichés dans la page active. La fonction « exportCsv » apparaît à l'intérieur de la fonction « imprimer » du menu de fonctions. (Voir : imprimer).

listDisplay : sert à personnaliser l'affichage et l'ordre d'apparition de propriétés sur la liste (voir option : configuration).

searchFields : sert à choisir les propriétés qui seront prises en compte au moment de faire une recherche sur le contenu de la liste. Cette propriété est utilisée avec la propriété « searchable » d'un champ.

sortFields : sert à choisir les propriétés qui seront prises en compte au moment de faire le classement du contenu de la liste. Cette propriété est utilisée avec la propriété « sortable » d'un champ.

hiddenFields : sert à choisir les propriétés qui ne seront pas affichées dans la liste. Noter que cette fonctionnalité cache un champ à l'affichage, elle ne l'efface pas de l'application.

personaliser des propriétés

Les exemples de personnalisation suivants montrent comment réaliser la configuration de base pour réussir une interface fonctionnelle et convivial.

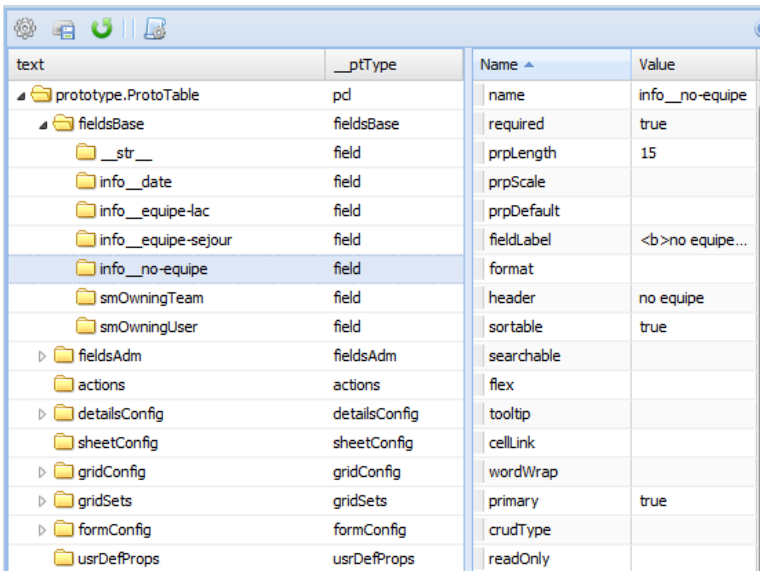


FIGURE 6.22: personnalisation des propriétés

La procédure de personnalisation est similaire aux autres personnalisations :

- Ouvrir le contenu du dossier « fieldsBase » (figure 6.4.2)en cliquant sur le triangle blanc situé à gauche du dossier.
- Sélectionner de la liste le champ à personnaliser.
- Modifier les valeurs.
- Ensuite, enregistrer les changements en cliquant sur le bouton « Mettre à jour le méta »

Les options plus communes dans la personnalisation des propriétés (champs) sont :

prpLength : valeur numérique qui définit le nombre des positions pour le champ. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

fieldLabel : chaîne de caractères qui définit l'étiquette du champ. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

header : chaîne de caractères qui définit l'entête des colonnes de la liste principale. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

sortable : valeur booléenne qui définit si le champ est utilisé pour classer le contenu de la liste principale. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

searchable : valeur booléenne qui définit si le champ est utilisé pour faire des recherches dans contenu de la liste principale. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

flex : valeur numérique (pourcentage %) qui définit la largeur des colonnes de la liste principale. Flex fait un calcul par rapport à la taille de l'écran et ajuste la largeur de la colonne en conséquence. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

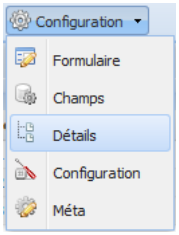


FIGURE 6.23: configuration des détails

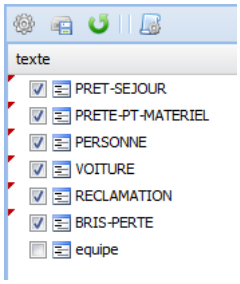


FIGURE 6.24: Activer les détails

tooltip : chaîne de caractères qui crée un message dans la forme d'une infobulle quand la souris passe sur le champ. Utilisé pour offrir plus des informations concernant le champ. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

wordWrap : valeur booléenne qui permet de visualiser le contenu du champ en plusieurs lignes. Propriété utilisée avec les propriétés de type texte ou de type string (quand la chaîne est très longue). Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

6.4.3 Configurer les détails

L'option « Détails » (figure 6.23) permet d'activer/désactiver la navigation vers les détails de la définition de le méta . Un détail est un MSI, donc une vue, construite sur un concept enfant. On navigue vers un concept enfant en cliquant sur un des détails disponibles dans l'entité parente. Les détails d'une entité sont accessibles à partir du menu de navigation (voir navigation). NOTE : Dans toutes le vues, tous les détails sont activés par défaut, mais il est possible de les activer ou les désactiver dans la définition du meta.

Pour activer ou désactiver les détails, crocher ou décrocher la case située à gauche du détail (figure 6.24). Ensuite, enregistrer les changements en cliquant sur le bouton « Mettre à jour la méta »

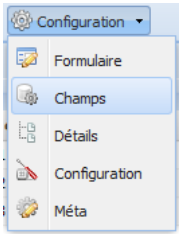


FIGURE 6.25: Configuration des propriétés

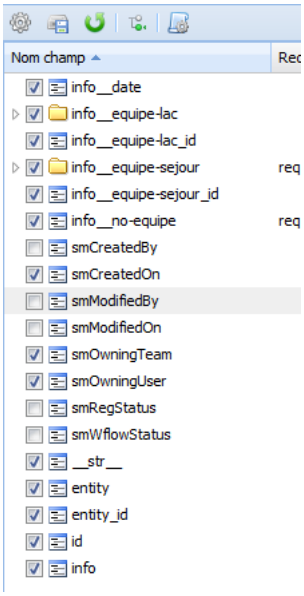


FIGURE 6.26: Activer les propriétés

6.4.4 Configurer les propriétés

L'option « propriétés » (figure 6.25) permet d'activer/désactiver les propriétés de la définition du méta. Toutes les propriétés des vues accessibles sont activées par défaut, mais il est possible de les activer ou les désactiver dans la définition du méta. Les absorptions des propriétés qui se trouvent dans les concepts qui sont une relation avec le concept principal (qui a le rôle de parent) se font aussi dans cette même fenêtre. Il est possible aussi d'activer des propriétés de type gestion (automatiques) comme la « date de la dernière modification », la « date de création » ou « modifié par ».

Pour activer ou désactiver les propriétés, crocher ou décrocher la case située à gauche des propriétés (figure 6.26). L'icône du dossier signifie que d'autres propriétés se trouvent à l'intérieur d'un concept parent. Pour montrer les propriétés d'un parent, cliquer sur le triangle blanc situé à gauche de l'icône dossier. Ensuite, enregistrer les changements en cliquant sur le bouton « Mettre à jour le méta ».

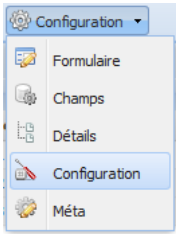


FIGURE 6.27: Configurer la liste

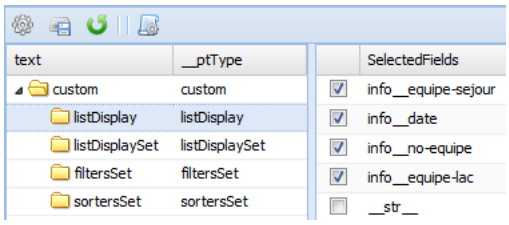


FIGURE 6.28: afficher les colonnes sur la liste

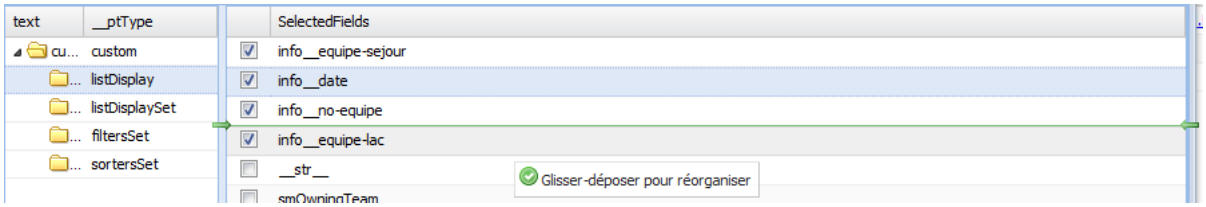


FIGURE 6.29: changer l'ordre des colonnes sur la liste

6.4.5 Configurer une liste

L'option « Configuration » (figure 6.27) donne accès au sous-ensemble de propriétés de personnalisation des listes. Cette option est visible pour les utilisateurs qui ont le droit de configurer l'application. Fondamentalement, cette option permet la personnalisation d'affichage de propriétés sur la liste principale.

Pour afficher ou cacher les propriétés, cliquer sur l'option « listDisplay » de la liste (figure 6.28). Ensuite, crocher ou décrocher la case située à gauche du nom du champ. Enregistrer les changements en cliquant sur le bouton « Mettre à jour le méta ».

Pour changer l'ordre d'apparition des colonnes sur la liste cliquer sur l'option « listDisplay » de la liste. Dans le panel de droit, glisser et déposer le champ vers le haut ou vers le bas pour changer l'ordre (figure 6.29). Ensuite, enregistrer les changements en cliquant sur le bouton « Mettre à jour le méta ».

6.4.6 Configuration des formulaires

La fenêtre de configuration de formulaire (figure 6.30) est organisée en trois grandes sections :

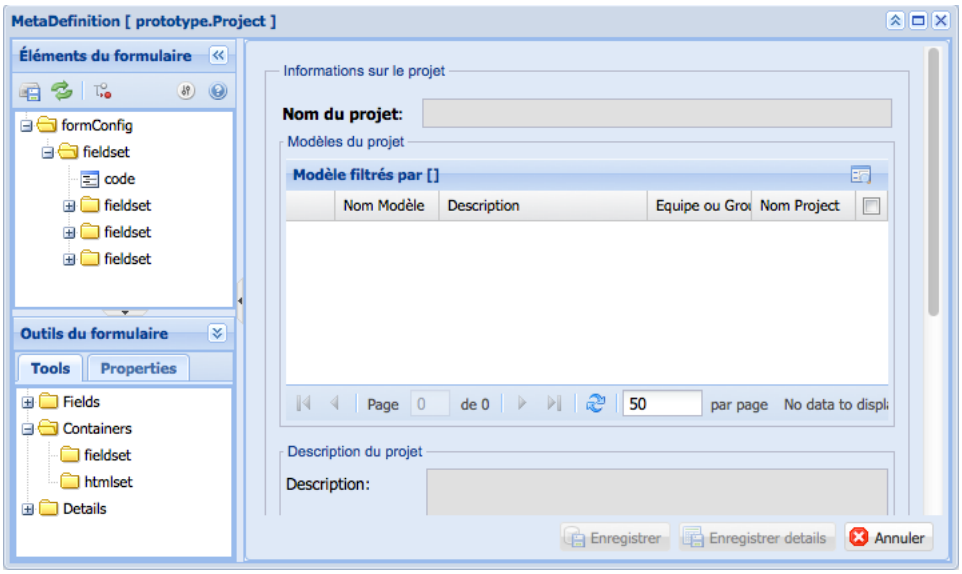


FIGURE 6.30: Configurer un formulaire

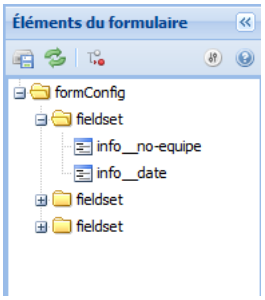


FIGURE 6.31: Éléments du formulaire

1. Les « éléments du formulaire » en haut du côté gauche de la fenêtre.
2. Les « outils du formulaire » en bas du côté gauche de la fenêtre.
3. L'« aperçu du formulaire » du côté droit de la fenêtre.

Les « éléments du formulaire »

La fenêtre des éléments du formulaire (figure 6.31) affiche dans une arborescence les propriétés qui forment le formulaire . À partir de cette fenêtre, il est possible de personnaliser l'affichage et l'ordre d'apparition de propriétés sur le formulaire. L'ordre d'apparition de propriétés est défini selon leur position dans l'arborescence. Il est possible aussi de regrouper les propriétés à l'intérieur des contenants de type « Fieldset » ou « HTMLset » pour indiquer des blocs d'information ou seulement pour mieux distribuer l'information sur l'écran.

La barre de configuration de formulaires présente les actions suivantes :

–  Enregistrer les modifications du formulaire.

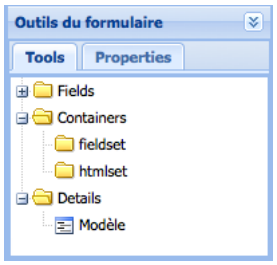


FIGURE 6.32: Les outils du formulaire

–  Rafraîchir l'aperçu du formulaire (côté droit de la fenêtre).

–  Supprimer un objet dans l'arborescence.

Les icônes dans l'arbre des éléments du formulaire sont :

–  Icône attribuée aux propriétés.

–  Icône attribuée aux contenants de type fieldset et HTML.

Les « outils du formulaire »

La fenêtre des outils du formulaire (figure 6.32) donne accès aux outils permettant de créer et de personnaliser un formulaire. Les outils sont organisés dans une arborescence où se trouvent les propriétés définies antérieurement pour le MSI, les contenants et les détails pour une vue. L'onglet des « propriétés » donne accès aux propriétés de personnalisation de l'objet sélectionné dans la fenêtre des éléments.

Propriétés (fields) Le dossier « Fields » contient la liste complète des propriétés pour le MSI sélectionné. Dans le cas d'une vue, les propriétés correspondent aux propriétés ou attributs des entités créées par l'utilisateur. On y trouve aussi des propriétés créées par l'application du prototypeur comme la date de la dernière modification, le nombre de l'équipe ou groupe à qui appartient le projet, etc.

Containers (Contenants) Le dossier « Containers » est composé de deux types de contenants qui servent à regrouper plusieurs propriétés dans le formulaire. Un contenant de type Fieldset crée un contour autour des propriétés qu'il entoure. Un contenant de type HTMLset, c'est un éditeur de texte HTML. Le contenant HTMLset s'utilise pour les propriétés de type de base « texte » qui stockent des chaînes de 65 535 caractères maximum.

Détails (Enfants) Le dossier « Détails » contient les détails (enfants) configurés pour le MSI. Quand un détail est ajouté au formulaire, le détail s'affiche dans la forme d'une liste. Il est préférable de placer les détails à l'intérieur d'un contenant.

« L'aperçu du formulaire »

L'espace réservé à « l'aperçu du formulaire » permet de prévisualiser les modifications appliquées au formulaire. Les modifications sont affichées en temps réel, cela veut dire que si on change l'ordre des propriétés ou que l'on regrouper plusieurs propriétés dans un fieldset, on voit dans l'espace de l'aperçu le résultat final quasi instantanément.

Personalisation des formulaires

Les exemples de personnalisation suivants montrent comment réaliser la configuration de base pour concevoir un formulaire fonctionnel et présentable. Par défaut les formulaires contiennent toutes les propriétés (champs) definies dans le MSI.

Pour éliminer un champ du formulaire

- Sélectionner le champ et cliquer sur le bouton supprimer élément du formulaire.
- Enregistrer les changements au formulaire en cliquant sur le bouton « Enregistrer formulaire ».

Pour ajouter un élément dans le formulaire

- Dans l'outil du formulaire, sélectionner l'onglet "outils",
- Sélectionner le dossier « Fields » pour visualiser la liste de propriétés disponibles (définis dans le MSI) .
- Sélectionner de la liste le champ à ajouter au fieldset.
- Glisser et déposer le champ à l'intérieur du fieldset.
- Enregistrer les changements au formulaire en cliquant sur le bouton « Enregistrer formulaire ».

Pour personnaliser un élément du formulaire

- Sélectionner l'élément à personnaliser de l'arborescence.
- Cliquer sur l'onglet « Properties » de la fenêtre Outils du formulaire (figure 6.33).
- En général les éléments les plus communs à personnaliser pour les contenants sont : title, fsLayout, labelAlign.

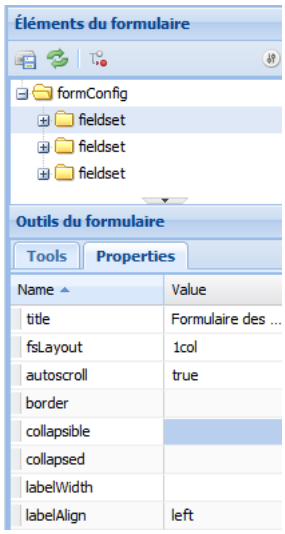


FIGURE 6.33: Propriétés des éléments du formulaire

title : ajoute un titre au contour du conteneur.

fsLayout : modifie la disposition des colonnes du fieldset. Pour avoir accès aux valeurs prédéfinies, cliquer deux fois sur le champ valeur de cette propriété.

labelAlign : Aligne les étiquettes de propriétés. Pour avoir accès aux valeurs prédéfinies, cliquer deux fois sur le champ valeur de cette propriété.

- En général les éléments les plus communs à personnaliser pour les propriétés (champ) sont : fieldLabel, tooltip et labelAlign.

  fieldLabel : chaîne de caractères qui est l'étiquette du champ. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

  tooltip : chaîne de caractères que crée un message dans la forme d'une infobulle quand la souris passe sur le champ. Utilisé pour offrir plus des informations concernant le champ. Pour y accéder cliquer deux fois sur le champ valeur de cette propriété.

- Editer les propriétés necesaires
- Enregistrer les changements au formulaire en cliquant sur le bouton « Enregistrer formulaire ».

Personnaliser les propriétés du formulaire

- Changer l'ordre des propriétés à l'intérieur du même contenant est très facile. Sélectionner le champ. Glisser et déposer le champ à l'emplacement désiré.
- Pour changer le champ d'un contenant à l'autre : sélectionner le champ, glisser et déposer le champ à l'intérieur du deuxième contenant.

Ajouter un contenant fieldset ou HTMLset à l'arborescence

- Cliquer sur l'onglet « Tools » de la fenêtre Outils du formulaire.
- Cliquer sur le plus « + » situé du côté gauche du dossier « Containers » pour visualiser la liste de contenants disponibles.
- Sélectionner dans la liste le type de contenant à ajouter à l'arborescence.
- Glisser et déposer le contenant à l'endroit désiré de l'arborescence. Il est possible d'insérer un contenant à l'intérieur d'un autre contenant.
- Enregistrer les changements au formulaire en cliquant sur le bouton « Enregistrer formulaire ».

Ajouter un détail à l'arborescence Les détails prennent automatiquement la forme d'une liste (comme la liste principale de l'application). La liste correspond à la définition d'une autre MSI lié par une relation parent-enfant. La liste n'est pas personnalisable à partir de la fenêtre formulaire.

- Cliquer sur l'onglet « Tools » de la fenêtre Outils du formulaire.
- Cliquer sur le plus « + » situé du côté gauche du dossier « Details » pour visualiser la liste de détails disponibles.
- Sélectionner dans la liste le détail à ajouter à l'arborescence.
- Glisser et déposer le contenant à l'endroit désiré de l'arborescence.
- Enregistrer les changements au formulaire en cliquant sur le bouton « Enregistrer formulaire ».

Chapitre 7

Démarche de Prototypage

Un des principes sur lequel reposent les méthodes dites Agile est l'interaction intense entre les clients, les utilisateurs et les développeurs, ce qui entraîne la validation précoce des spécifications. Pour nous, la démarche de conception doit considérer le prototypage, comme un «processus» (Floyd, 1984) [1]. Le prototypage doit être en soi une phase bien définie au sein du cycle de vie du développement du logiciel. Les modèles peuvent présenter toutes sortes de défauts, mais, quand les client sont réellement assis en face d'un prototype et travaillent avec lui, les défauts deviennent vraiment évidents : à la fois en terme d'erreurs et en terme d'exigences mal comprises. La démarche que nous proposons est basée sur les points suivants :

1. Interpréter le réel perçu
2. Concevoir le modèle conceptuel (objet-relationnel) de données.
3. Entrer le modèle dans l'application du prototypeur
    a) Entrer ou modifier les concepts, propriétés et relations
    b) Afficher la vue graphique
    c) Générer les formulaires pour chaque table du modèle
    d) Alimenter des valeurs dans les entités vanille.
4. Concevoir les vues d'utilisateur
    a) Choisir le concept de base
    b) Choisir les zooms (lookup) et les propriétés absorbées
    c) Composer la navigation parent-enfant
    d) Composer les formulaires d'affichage
5. Valider avec l'utilisateur
6. Vérifier et retourner au pas #1 au besoin.

1. cite par [45]

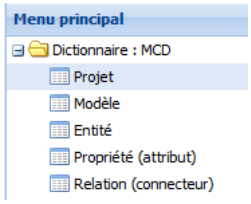


FIGURE 7.1: Menu de création du prototype

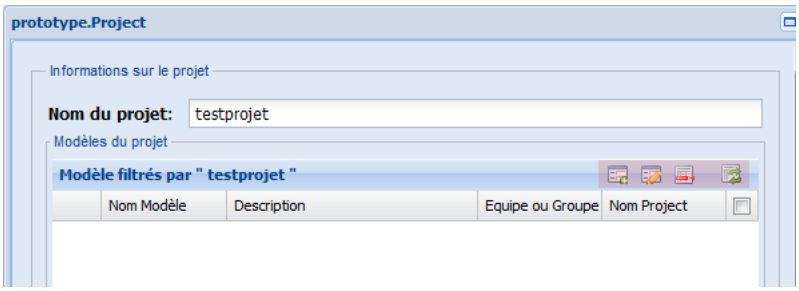


FIGURE 7.2: formulaire des projets

L'interprétation du réel perçu et la conception sont des processus mentaux d'abstraction [46]. L'outil commence à être formellement utilisé à partir de l'étape #3. Les analystes alimentent un dictionnaire avec des modèles des données conceptuels (normalisés) basés sur un formalisme relationnel.

7.1 Créer un projet

La première étape pour construire un prototype consiste à créer un nouveau projet (figure 7.1) pour le réel perçu concerné par le projet d'informatisation. Le modèle de données doit être créé à l'intérieur d'un projet existant. Un projet peut contenir un ou plusieurs modèles (figure 7.2). Le recours à un découpage en modèles est un choix de l'analyste en charge de la réalisation du prototype.

7.2 Créer un modèle

La deuxième étape pour construire un prototype consiste à créer un modèle conceptuel de données qui décrit d'une manière commune de multiples usages du SI qui font l'objet d'une spécification et d'une validation sous la forme de MSI (figure 7.3).

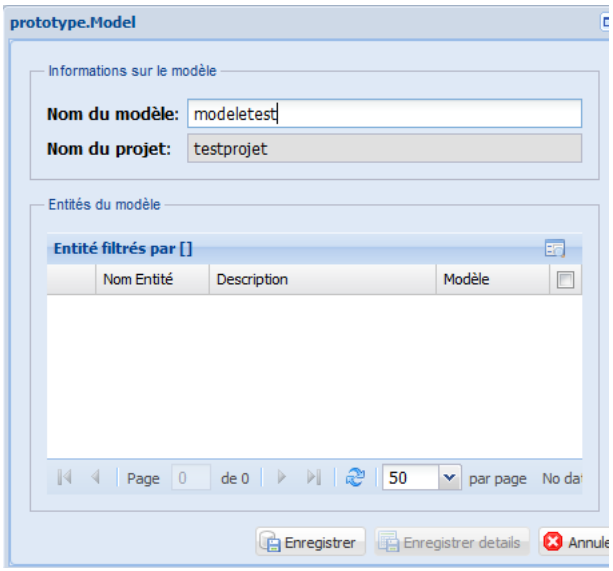


FIGURE 7.3: formulaire des modèles

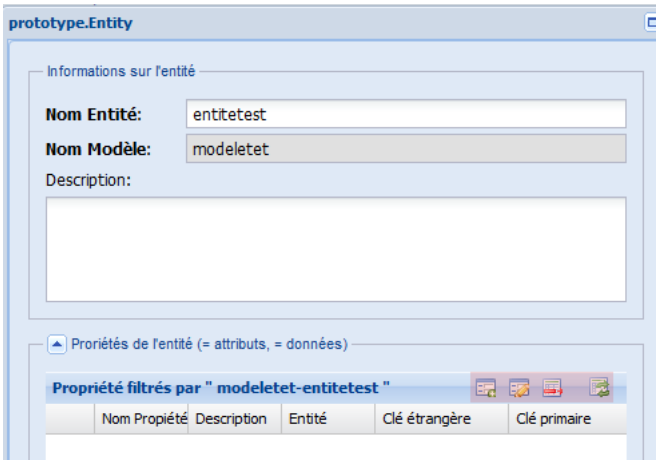


FIGURE 7.4: formulaire des concepts et propriétés

7.3 Créer les concepts

La troisième étape pour construire un prototype consiste à créer les concepts[2] du modèle conceptuel de données. Le concept est la perception d'une « chose »[3] dans la réalité. On doit créer les entités (en fait concepts) à l'intérieur du modèle créé à l'étape antérieure (figure 7.4). Un modèle de données peut contenir plusieurs entités (concepts), mais une entité (concept) ne peut appartenir qu'à un seul modèle.

2. Pour des raisons historiques les menus et les formulaires affichent encore le nom « Entité ce qui sera changé dans la prochaine version.
3. On n'utilise pas le mot « objet » pour ne confondre avec la terminologie OOP

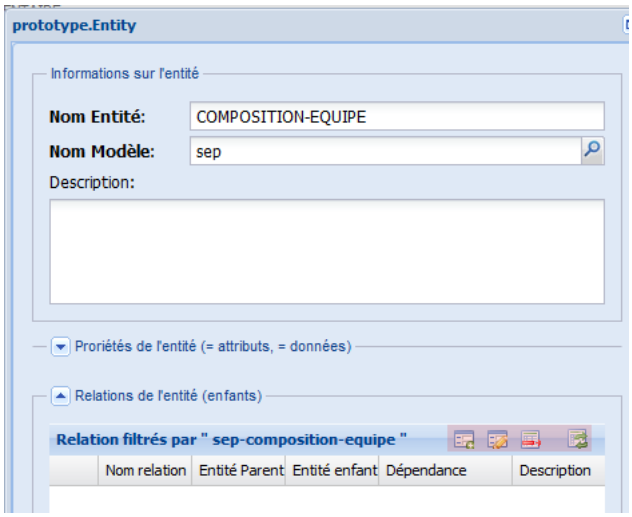


FIGURE 7.5: formulaire des concepts et relations

Le formulaire de concept comporte deux enfants (figure 7.5) : Les propriétés et les relations [4].

7.4 Créer une propriété

La quatrième étape pour construire un prototype consiste à créer les propriétés des entités (concepts) du modèle conceptuel de données (figure 7.6). Dans notre vocabulaire les propriétés sont les éléments de données qui portent les valeurs. Par exemple le « nom du client » , « le numéro de facture », etc...

On doit créer les propriétés à l'intérieur de chacune des entités créées à l'étape antérieure.

Les attributs des propriétés sont les suivantes :

- le « nom de la propriété » (information obligatoire)
- le « nom de l'entité » (information obligatoire).
- la « description de l'entité » (information optionnelle).
- la « clé primaire » indique si la propriété participe la clé sémantique (celle que connait l'utilisateur) de l'entité. Quand on croche cette case, la case de « Requis » est crochée automatiquement (l'identifiant est en effet obligatoire).
- la case « Requis » si la propriété n'est pas une clé primaire, mais que la propriété est obligatoire.
- La « Clé étrangère » est disponible en lecture seulement, indique que c'est un attribut de type relation, elle est cochée automatiquement par l'application lors de la spécification d'une relation.

4. qui sont en même temps un type special de propriété comme discuté dans le chapitre précédent

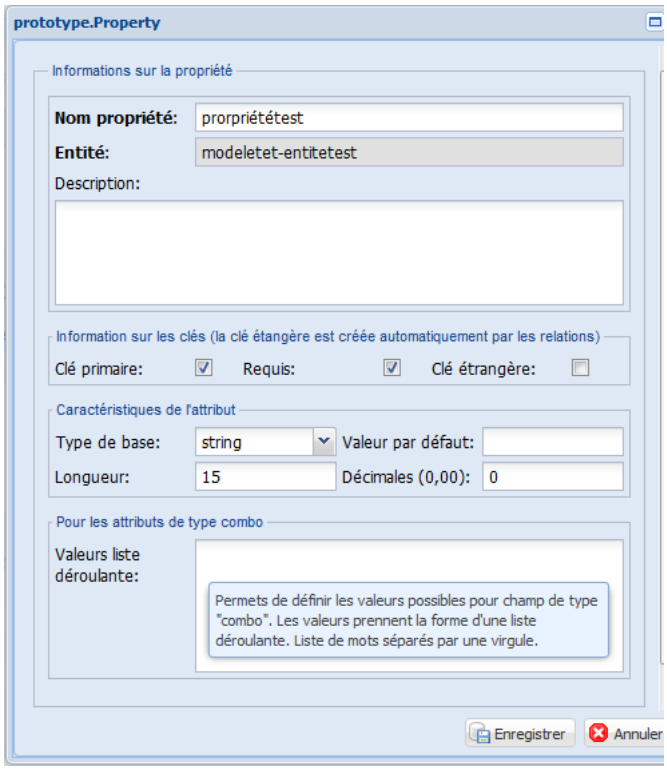


FIGURE 7.6: formulaire des propriétés

- le « Type de base » : information optionnelle par défaut « string », ce qui permet à l'utilisateur de ne pas se préoccuper de ce « détail » qui pour lui va de soi, mais qui du point de vue de la programmation est essentiel. Si le choix de type de base est décimal, on remplit le champ « Décimales ». Si le choix de type de base est combo (liste de valeurs), on remplit le champ « Valeurs de la liste déroulante » et le champ « Valeur par défaut ».

- la « Longueur » (information optionnelle) indique la longueur maximal d'un texte ou d'un numérique.

- la « Valeur par défaut » (information optionnelle).

- les « Décimales » (information optionnelle). Ce champ est utilisé en combinaison avec le type de base décimal, il précise le nombre de positions après la virgule pour un chiffre.

- les « Valeurs liste déroulante » (information optionnelle). Ce champ est utilisé en combinaison avec le type de base combo. Les mots doivent être séparés par des virgules sans espaces en blanc. Exemple : QC,ON,NS,NB,BC...

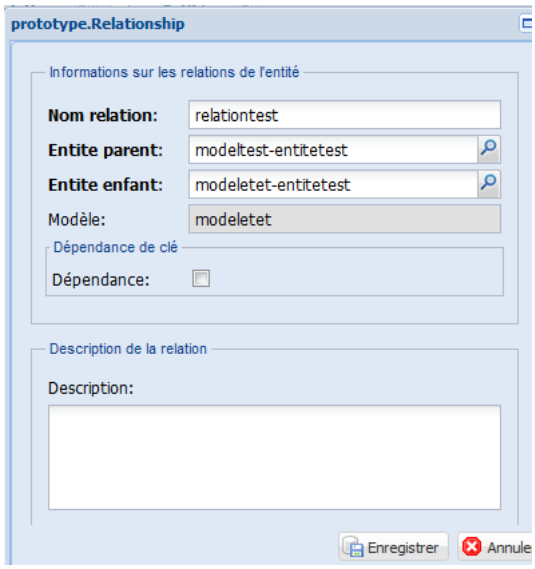


FIGURE 7.7: formulaire des relations

7.5 Créer une relation

La cinquième étape pour construire un prototype consiste à créer les relations entre les entités du modèle conceptuel de données. Les relations peuvent être créées à partir de l'entité enfant vers l'entité parente ou vice-versa (figure 7.7). Les clés étrangères de nature sémantique (propriétés) sont générées automatiquement par l'application du prototypeur au moment de créer la relation entre deux entités.

Les attributs des relations sont les suivantes :

- le « nom de la relation » (information obligatoire). Comme bonne pratique, composez les noms de vos relations en commençant par le nom de l'entité enfant suivi du nom de l'entité parent. Les deux noms séparés par un trait d'union, exemple séjour-chalet.
- le « nom de l'entité parent » (information obligatoire). Cliquez sur le bouton pour sélectionner l'entité de la liste.
- le « nom de l'entité enfant » (information obligatoire). Cliquez sur le bouton pour sélectionner l'entité de la liste.
- Crocher la case de la « dépendance de clé » si la connectivité du côté enfant est dépendante : 1,1. Si la connectivité est 1,1, ne pas crocher cette case. La dépendance de clé est gardée pour compatibilité avec le modèle relationnel et donne lieu aux types abstraits de relation (absorption, composition, ...)
- la « description de la relation » (information optionnelle).

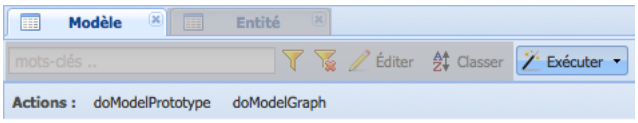


FIGURE 7.8: générer le modèle graphique

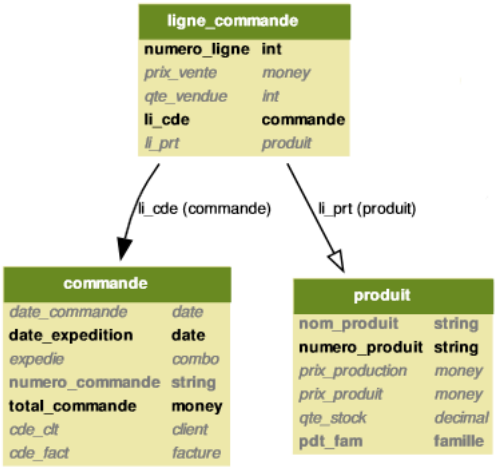


FIGURE 7.9: exemple de modèle graphique

7.6 générer le modèle graphique

L'action « doModelGraph » génère le modèle conceptuel graphique pour un modèle sélectionné. À partir de l'onglet « Modèle » (figure 7.8).

- Cliquer sur la ligne de la grille principale qui correspond au modèle à générer.
- Cliquer sur le bouton « Exécuter » du menu des fonctions.
- Cliquer sur l'option « doModelGraph ». Un message de confirmation de l'opération sera affiché sur l'interface (Voir la barre de messages pour plus de détails).
- Le modèle graphique généré est ouvert dans un nouvel onglet du fureteur. Le format de sortie de la graphique est un fichier PDF.

Le formalisme utilisé dans la représentation actuel est celui du modèle relationnel-objet décrit précédemment :

- Clé primaire : la police du texte est grasse et sa couleur est noire. Exemples dans la figure : numéro_ligne, numéro_produit et date_expédition.
- Clé étrangère : la police du texte est italique et sa couleur est gris pâle. Le nom de son entité enfant apparaît du côté droit du nom de la clé. Exemple : li_prt - produit ou cde_fact - facture.
- Clé étrangère dépendante (obligatoire) : la police du texte est grasse et sa couleur est noire. Le nom de son entité enfant apparaît du côté droit du nom de la clé. Exemple : li_cde - commande.

- Clé étrangère requise (obligatoire) : la police du texte est grasse et sa couleur est gris pâle. Le nom de son entité enfant apparaît du côté droit du nom de la clé. Exemple : pdt_fam - famille.
- Propriété : la police du texte est en italique et sa couleur est gris pâle. Exemples dans la figure : prix_vente, qte_vendue et prix_production.
- Propriété requise (obligatoire) : la police du texte est grasse et sa couleur est gris pâle. Exemples dans la figure : nom_produit et numéro_commande.
- Relation dépendante (obligatoire) : la flèche est pleine.
- Entité parent : la flèche pointe vers l'entité parent.

7.7 Génération du prototype

L'objectif de notre projet est la génération de prototypes. Elle commence par la création automatique des vues d'utilisateurs (MSI) basées sur les modèles conceptuels de données qui ont été fournis dans les étapes précédentes. Le système est capable de générer une vue d'utilisateur à partir de chaque « Concept » du modèle. Cette vue peut être complétée pour l'adapter au besoin de l'utilisateur directement dans l'environnement de l'application de prototypage.

Les nouvelles vues sont générées automatiquement comme points d'entrée à l'intérieur de l'arborescence « AutoMenu » - « ProtoOptions ». Le nom par défaut de la vue est « nom du modèle . nom de la vue ». Les vues des prototypes antérieures sont aussi affichées avec les vues récemment créées.

La génération des vues de prototype se fait soit à partir du modèle, soit à partir d'un concept spécifique. Quand elle est faite à partir du modèle, chacun des concepts donne lieu à une vue par défaut. Quand elle est fait directement sur un concept, l'utilisateur (analyste) doit fournir un nom pour la vue. Cela permet de générer plusieurs vues sur un même concept.

Les titre des vues sont composés du nom du modèle suivi d'un tiret (-) et du nom de l'entité qu'elle représente. Les vues dans l'arborescence « AutoMenu » - « ProtoOptions » sont de vues générées automatiquement, si vous effectuez des modifications sur ces vues, la prochaine fois que le menu principal sera rafraîchi, les vues retourneront à leur valeur par défaut. Pour garder les modifications des vues, il est nécessaire de créer un nouveau dossier sur l'arborescence du menu principal.

Pour classer le vues dans le menu principal de la application :

- Cliquer sur le bouton «Nouveau dossier » du menu principal. Entrer le nom du nouveau dossier, par exemple le nom du projet. Le nouveau dossier apparaîtra à la fin de la liste des vues générées dans « AutoMenu » - « ProtoOptions ».
- Procéder par glisser-déposer une vue à la fois à l'intérieur de ce dossier sur les vues appartenant à ce projet.

- Quand toutes les vues sont à l'intérieur du dossier, glisser et déposer le dossier vers l'arborescence du menu principal. Par exemple, après la composante relation. S'assurer que le dossier se retrouve à l'extérieur de l'arborescence « AutoMenu ».
- Sélectionner « AutoMenu » et cliquer sur le bouton « Supprimer nœud» du menu principal. Ceci efface l'arborescence d'« AutoMenu » (la suppression n'est pas définitive, pour récupérer cette arborescence cliquer sur le bouton réinitialiser du menu principal).
- Pour terminer, cliquer sur le bouton « Enregistrer menu » du menu principal. Important ! si on n'enregistre pas les modifications au menu, elles sont perdues à la fermeture de l'application du prototypeur.

Conclusion

Le processus de conception des SI est un processus long et complexe qui se traduit par de nombreux échecs. Le prototypage informatique, a été proposé depuis longtemps, à l'instar des ateliers de génie logiciel (CASE Computer Assisted System Engeniering), de la génération de code et de plusieurs autres alternatives pour améliorer ce processus. Une des sources de ces difficultés a souvent été identifiée comme la difficulté de définir les spécifications initiales, ce qui invite à améliorer cette étape cruciale. Construire un prototype y contribue directement et plusieurs méthodes y ont recouru ou se prêtent plus ou moins bien à son utilisation. Plusieurs avantages ont été avancés pour le recours à cette technique qui toutes concernent la communication « Client - Concepteur » et « Concepteur - Développeur » pour une amélioration de la qualité du SI :

- Le prototype assure que les experts de domaine et les informaticiens parlent le même langage.
- Limitation des itérations parce qu'il existe un entente validée sur les besoins validé par un prototype.
- Participation accrue de l'utilisateur
- Meilleure qualité des exigences et des spécifications fournies aux développeurs
- La détermination précoce de ce que l'utilisateur veut vraiment peut entraîner un solution plus rapide et moins coûteuse
- Offre une meilleure et plus complète informations des spécifications
- Empêche de nombreux malentendus
- Peut conduire à un solution de plus grande qualité

La conception guidée par le modèle (CGM) est une approche de construction des SI qui vise un passage plus direct à partir de la modélisation vers la production des SI.C'est dans ce contexte que nous avons entrepris ce travail : comment la CGM peut être utilisée pour construire des prototypes de SI transactionnels (SIT) et comment insérer le mieux possible le prototypage dans le cycle de développement.

Après une première étape conceptuelle qui a consisté à expliciter la problématique de la spécification et de la construction des SI nous avons réalisé un outil de prototypage qui devait

nécessiter en entrée des informations minimales centrées sur l'essentiel : le modèle de conceptuel de domaine (MCD) et les modèles de spécification d'interface (MSI). Nous envisagions le recours à un prototype comme l'approche "idéale" pour la conception progressive des SI.

Au cours de la réalisation nous avons pu vérifier que la démarche de construction des prototypes peut s'accorder avec une démarche de conception progressive des SI. D'une part, tel qu'énoncé par la méthode Datarun, le point de départ pour la construction de un SI est la recherche de ce qui est invariable [5] dans l'organisation c'est à dire les données. Les utilisateurs, en tant que consommateur d'information s'attendent à créer et à exploiter des données, et ils les manipulent au moyen des interfaces construites à partir de ces mêmes données. Le deux besoins fondamentaux sont donc, un modèle conceptuel des données (MCD) et un modèle de spécification d'interface (MSI). D'autre part, on constate que les exigences des utilisateurs sont inscrites dans un contexte (SIO), et que dans cet contexte il y a des règles et des connaissances qui sont implicites. En d'autres termes l'utilisateur prend pour acquis une série de considérations qui pour lui sont évidentes. La travail de l'analyste est de rendre explicite les connaissances nécessaires au code qu'il écrit. Il devient alors indispensable d'avoir un mécanisme de communication qui permet de mettre en évidence les aspects implicites du contexte.

Nous avons été conduits à mettre en évidence le passage du monde de l'utilisateur à celui du code dans l'ordinateur qui est une machine abstraite à connaissance finie : ceci est réalisé par le passage des identifiants connus par l'utilisateur (que l'on appelle sémantique de son point de vue) aux pointeurs utilisés par le code, deux techniques par ailleurs bien connues. Nous avons ainsi mieux compris le délicat passage du monde réel au monde abstrait de l'ordinateur dans lequel tout est une « chaîne de bits ». Le mot sémantique s'est révélé un véritable piège car la définition que nous avons retenue, ce qui établit la correspondance entre une chose et sa réalité s'applique différemment dans ces deux mondes, dans le réel on a un identifiant, dans le système informatique on a des références (souvent appelés dans les sgbd). L'outil que nous avons réalisé est construit autour de la mise en œuvre de cette problématique.

Grâce à l'expérience accumulée dans les disciplines informatique, notamment le recours à un méta-modèle et à la conception guidée par modèles (CGM), en exploitant l'immense patrimoine de code libre, nous avons réussi à implémenter l'architecture pour construire un outil de construction automatique du code simple et puissant. Il devient alors possible de construire concrètement, d'une manière dynamique, un prototype du système d'information. La vitesse de développement, la souplesse de l'architecture produite et les techniques de réutilisation des composantes permettent de livrer des versions successives des prototypes de SI au fur et à mesure que l'utilisateur explicite ses exigences. Ce qui répond à notre question de départ de : Comment la CGM peut être utilisée pour construire des prototypes de SI transactionnels (SIT).

5. Si on s'accorde à qu'il n'y a pas de choses invariables, au moins les données sont les structures les plus permanentes.

À l'aide de l'outil de prototypage l'utilisateur peut évaluer les conséquences de changements dans le designs et manipuler un système fonctionnel pour constater si son idée est bien implémentée. Le prototype construit par l'analyste assure, après quelques itérations, un point de départ solide qui devient une entente de besoins et par la suite on peut continuer avec le développement du système informatique proprement dit. Ce qui répond à notre objectif de départ qui est de mieux orienter l'action de spécification des exigences dans la phase initiale de conception « Communication Client - Concepteur » et dans le début de la phase de développement « Communication Concepteur - Développeur » en utilisant des artefacts de prototypage.

La construction de l'outil de prototypage a impliqué de notre part de nombreuses étapes (spirales). La construction du méta-modèle pour héberger les modèles conceptuels et les modèles d'interface ont été un processus long et chaotique qui a débouché sur une clarification des concepts. Progressivement nous avons pu faire cadrer l'architecture avec une vision épurée des concepts de donnée et connaissance en vue de produire de l'information (voir le chapitre 2). La version actuelle du méta-modèle est le résumé des longues journées de discussion constructive, le but était de de trouver une solution finalement simple mais qui n'était pas évidente. Nous sommes ainsi arrivé à maîtriser les connaissances nécessaires pour construire un prototype que se construit lui même. L'outil de prototypage a été construit en utilisant le même outil (bootstrapping : le rêve des programmeurs).

Une des dernières caractéristiques ajoutée au prototypeur : la création le dynamique des structures souples pour enregistrer les modèles en « brouillon » qui était demandée par les premiers utilisateurs du prototypeur ouvre de nouvelles perspectives. La première version directement construite sur Django avait l'inconvénient d'être « statique », la modification du modèle impliquait une reconstruction (incluant un nouveau design) de l'application construite par l'analyste. Pour résoudre ce problème nous avons dû ne plus nous ancrer directement dans la couche abstraite d'accès à la base de données relationnelle. En conséquence, nous avons a élaboré une solution à partir du patron EAV [6] et des types de données objet analogues à un triplet à la base du web sémantique.

Le recours au prototypage classique a cependant des inconvénients :

- L'utilisateur est conduit à penser qu'un prototype est un système final, ce qui conduit à des attentes irréalistes qui sont déçues
- Attendre du prototype les performances de système final
- S'attacher à fonctionnalités incluses dans le prototype et retirées dans le système final

Toutefois, le principal inconvénient est l'abandon du prototype qui ne sert plus à l'évolution du SI au cours de sa vie. Le prototype est par définition un outil de documentation de la planification initiale indépendant de la réalisation, donc, jamais à jour par la suite avec les

6. EAV : acronyme pour Entity Attribute Value

systèmes en production. Les prototypes sont considérés comme des produits jetables, ils ont pour seul rôle la spécification concrète des spécifications du système en construction[7]. On pourrait argumenter qu'il suffit de documenter la réalisation de telle manière que les utilisateurs avec les analystes puissent piloter l'évolution du SI, l'expérience a montré que ce n'est pas une bonne solution. Il faudrait que toute démarche d'évolution du système en production passe obligatoirement par l'étape de prototypage ce qui implique la mise à jour de modèles. C'est précisément la voie qui est concrètement ouverte par notre réalisation, il devient possible d'envisager que toute modification au cours de la vie d'un SI puisse être effectuée à partir du modèle du domaine qui est l'input du prototypeur, qui devient alors le SI lui-même. Soit la modification concerne les modèle et alors elle peut être faite directement en input et le SI s'ajuste automatiquement, soit elle porte sur des fonctionnalités techniques (briques constitutives) qui sont arrimées au modèle objet-relationnel stocké dans les métadonnées. C'est cette piste que nous espérons pouvoir explorer à partir de ce travail qui ouvre une voie concrète à notre objectif implicite à long terme initial : réaliser un SI à partir de spécifications déclaratives minimales.

Ainsi, en élaborant cette solution, nous avons eu l'opportunité de nous inspirer de la logique de prédicats. C'est un monde fascinant que permet la construction de nouvelles connaissances par inférence à partir de données et de connaissances initiales (voir exemple 2.2). La logique de prédicats est à la base de la web sémantique qui vise à aider l'émergence de nouvelles connaissances en s'appuyant sur les connaissances déjà existants sur Internet[34]. Notre architecture est prête pour évoluer afin d'exploiter la richesse des prédicats sur la web sémantique.

Enfin l'ensemble du code produit est retourné à la communauté qui nous a tant apporté, il est sous licence GPLv2 ou suivantes, disponibles (ainsi que toutes nos errances) sur github à l'adresse : `https://github.com/certae/ProtoExt`

7. car la mise à jour du prototype implique des efforts supplémentaires

Annexe A

Description du code de projet

Langages de programmation et cadre d'applications

Le code source du projet est écrit essentiellement dans différents outils informatiques, nous présentons succinctement ce que nous avons écrit. Le code produit est retourné à la communauté qui nous a tant apporté, il est sous licence GPLv2 ou suivantes, disponibles (ainsi que toutes nos errances) sur github à l'adresse : `https://github.com/certae/ProtoExt`

Python

Python est le langage de base de **Django** [1] il est utilisé pour le support de BackEnd. Le BackEnd est la couche de services de l'application. La plupart de processus s'exécutent dans cette couche. Django est un cadre de logiciel (framework) qui fournit une vision d'architecture et une série de librairies génériques pour différents traitements nécessaires dans le développement d'une application web (Voir : 4.4.2) .

Django fournit un librairie "Admin" qui est un CRUD générique sur les concepts [2] définis, qui sont interprétés directement par l'ORM. Le ORM il assure le lien entre la base de données relationnelle et le traitement objet. Il permet la définition et la manipulation des données directement comme des objets en python. L'ORM autorise des exceptions pour sauter la couche d'abstraction objet et aller exécuter directement de requêtes SQL. Nous avons choisi de toujours rester dans la couche objet et il n'y aucune requête SQL pour la manipulation des données [3]. Rester dans la couche objet amène l'indépendance total des fournisseur de base de données.

Le support de métamodèle, la création et l'interprétation des métadonnées ont dû être codés dans ce projet. Django ne fournit pas ces concepts. Pourtant, le fait qu'il soit open source

1. https ://www.djangoproject.com/
2. Django appel "model" ce qui on appelle concept.
3. Bien que parfois utiliser SQL puisse être le chemin plus facile

nous a permis d'étudier son fonctionnement interne et apprendre la façon dont il traite des situations particulières.

Js

Js est la base de **ExtJs** [4], il est utilisé pour le support de FrontEnd. (Voir : 4.4.1). ExtJs fournit un ensemble de composantes "prêtes à utiliser" pour la manipulation de l'information directement dans le navigateur (FrontEnd). Sencha est aussi le créateur d'autres cadres d'applications qui sont spécialisés pour les appareils portables [5] fondés sur la même philosophie et avec certain niveau de compatibilité [6]. Les composants de ExtJs sont génériques et ils offrent une fonctionnalité de base. Javascript (js) n'est pas à proprement dit un langage OO, mais il dispose quand-même de mécanismes qui permettent "spécialiser " [7] les widgets de base pour obtenir ce dont on a besoin. Au cours du projet, nous avons spécialisé des widgets génériques tels que les grids , textbox , containers et autres pour "comprendre" directement les métadonnées fournies au BackEnd. Encore un fois, le fait qu'il soit open source nous a permis d'étudier son fonctionnement interne et apprendre comme implémenter les fonctionnalités nécessaires pour le prototypeur.

Dot

Il y a un troisième langage utilisé pour générer les diagrammes de l'application. dot est un langage de spécification graphique [8] à partir d'un définition de haut niveau.

Le prototypeur se résume au traitement de métadonnées pour décrire et automatiser l'implémentation d'un système d'information. La définition d'un modèle passe par le design des modèles que sont par la suite interprétés. Pourtant interprétation ne se fait pas sur les modèles, elle se fait plutôt sur la définition du modèle, c'est à dire ses métadonnées. Donc, le modèle n'est pas le graphique c'est la construction représentée par les métadonnées. L'approche que nous avons choisie est d'enregistrer les donénes du modèle et par la suite générer le graphique. Pour le faire, on a utilisé le langage "Dot".

On peut peut-être trouver pervers d'utiliser deux représentations différentes (textuelle et graphique) pour le même modèle sous-jacent. Mais il n'y en a qu'une, c'est la définition du modèle qu'on utilise pour exécuter, et qui devient un graphique pour la compréhension humaine. Il est vrai que la disposition du modèle peut aussi représenter des idées qui sont importantes pour la compréhension, mais il peut aussi exister une sémantique claire dans la

4. http ://www.sencha.com/products/extjs
5. http ://www.sencha.com/products/touch (téléphones et tablettes)
6. Cela permet un autre chemin de développement pour la créations d'outre type de clients (FrontEnd) qui utilisent les mêmes services de BackEnd.
7. Parler d'héritage en js risque d'énerver au puristes OO
8. http ://www.graphviz.org/

génération du modèle. Ce point a été longuement discuté et je vais expliquer mon point de vue [9].

Un modèle est une représentation partielle suivant un certain point de vue de la réalité, les modèles qui nous concernent sont souvent conçus graphiquement, mais ils doivent être exprimés comme un artefact particulier qui est un logiciel au sein d'un SI. Cependant, nous préférons généralement examiner nos modèles dans une représentation graphique. Par exemple UML emploie neuf formalismes différents pour visualiser les différents points de vue d'un système. L'utilisation d'un diagramme pour représenter un modèle a un certain nombre d'avantages. Lorsque nous examinons la représentation graphique d'un modèle que nous utilisons notre appareil cognitif visuel qui a des millions d'années d'évolution avantage sur nos capacités de lecture des textes.

Les concepteurs créent généralement leurs diagrammes de modèle à l'aide d'un éditeur de dessin. La distance sémantique entre la représentation du modèle graphique de l'éditeur et l'artefact logiciel sous-jacent peut varier énormément. Certains outils tels que Visio ou Draw sont des aides de dessin pur, d'autres comme OpenModelSphere ou ArgoUML peuvent offrir un aller-retour conceptuel. Cependant, tous les éditeurs de dessin exigent la manipulation et la mise en position de dessins des formes sur le canevas. Pourtant, les efforts et les compétences de coordination motrice nécessaires pour cette activité n'ont rien à voir avec le système final à différence des modèles de génie architectural ou mécanique dont l'apparence du diagramme et liée directement au produit final. L'activité de dessin est toutefois une tâche créative qui fournit une rétroaction immédiate et les analystes se concentrent très souvent sur la fourniture d'une belle image, plutôt que d'un design efficace.

La modélisation déclarative est aussi très malléable, la structure visuelle existant n'empêche pas des changements drastiques, ni gaspillage d'efforts sur l'arrangement ordonné de nœuds d'un graphe, que deviens une barrière psychologique contre le ré-factoring massif de la conception. À différence du modèle graphique, la modification sur des modèles textuelles (des métadonnées) est naturellement liée aux activités de ré-factoring, de contrôle de versions (CVS), de génération automatique, et d'extraction des métriques entre autres.

Aujourd'hui la modélisation déclarative utilise la puissance des algorithmes de dessin automatiques de graphes[4] et ont suffisamment avancé dans la façon de placer les nœuds et le routage optimal des relations. Nous pouvons donc concevoir des modèles en utilisant une représentation textuelle déclarative et ensuite voir, publier et partager sous forme graphique.

La modélisation déclarative n'est clairement pas une panacée. Notre outil de prototypage actuel souligne le point, on ne peut pas replacer les concepts pour exprimer l'importance qui entraîne, les lignes de relation se chevauchant et peuvent porté à confusion, etc [10]. Toutefois, je

9. Références : [5][4][80][7]
10. Par ailleurs, nous n'avons pas l'expertise pour optimiser la génération graphique.

crois que la construction et l'adoption d'outils de modélisation déclarative permettra faciliter la tache de la spécification déclarative de systèmes d'information.

Style d'architecture

Avant de commencer à expliquer le code il est important de se situer dans la fonctionnalité. Traditionnellement les différents patrons d'architecture dont MVC servent à définir la séparation de fonctions et l'organisation de sources.

MVC, est l'un des patron d'architecture plus connues, Django implémente un variante de MVC qu'ils appelle MVT (Model View Template)[11]. ExtJs permet aussi d'implémenter sa propre compréhension de MVC surtout orientée à l'organisation du code.

Le principe de MVC[79] est le suivant :

- Le modèle manipule l'information
- La vue s'occupe de la présentation
- Le contrôleur répond aux demandes de l'utilisateur et manipule le modèle

Un variation de cet principe dans laquelle la vue n'a pas de lien avec le modèle, et c'est le contrôleur qui fait toujours l'interface entre le deux. Notre projet est plus proche de cette vision du patron. Voici notre interprétation MVC.

- Le modèle manipule l'information. Le modèle est représenté par des classes qui sont écrites en python et gérées par l'ORM Django.
- Les vues sont conçues dynamiquement à partir de la définition de MSI, la définition est envoyée au client par le biais du contrôleur. Les vues sont matérialisées à partir des composants désignés sur ExtJs.
- Le contrôleur fait tout la logique d'interprétation du métamodèle et l'exécution dynamique. Pour l'aspect communication nous nous servons des composants spécifiques de Django et ExtJs.

Le code

À la base du code, on peut trouver trois dossiers principaux :

- src : contient le code python (BackEnd)
- static : contient le code js (FrontEnd)
- templates : contient les gabarits HTML qui hébergent le code de FrontEnd

11. `https://docs.djangoproject.com/en/dev/faq/general/django-appears-to-be-a-mvc-framework-but-you-call-the-contr`

“/src” : code python (BackEnd)

Le dossier /src suit les normes standard de Django. Il contient les fichiert :

- settings.py avec les paramètres de l'application dont :
  - Le fournisseur de base de données
  - La localisation de Django.
  - La localisation de ExtJs
- manage.py point d'exécution de Django.
- urls.py schema des Urls et la corrélation avec les fonctions qui fournissent les services.

Des dossier avec les applications (modules) qui supportent l'application “prototypeur” :

- /src/protoLib contient les services de soutien a la gestion générique des vues.
- /src/prototype contient les services spécifiques de prototypage

“/src/protoLib”

Voici un liste des services génériques exécutés lors de l'utilisation de l'application :

/src/protoLib/fields.py

permet de gérer les propriétés dynamiques du prototypage.

/src/protoLib/fields.py:

```
 33: class JSONDict(dict):
 35:     def __repr__(self):
 39: class JSONAwareQuerySet(models.query.QuerySet):
 40:     def __init__(self, json_fields = [], *args, *kwargs):
 44:     def _filter_or_exclude(self, negate, *args, *kwargs):
 71:     def _evaluate_json_lookup(self, item, lookup, value):
 88:         def _getattr(obj, key):
112:     def count(self):
115:     def all(self):
118:     def order_by(self, *args, *kwargs):
121:     def _clone(self, *args, *kwargs):
127: class JSONAwareManager(models.Manager):
128:     def __init__(self, json_fields = [], *args, *kwargs):
132:     def get_query_set(self):
137: class JSONField(models.TextField):
140:     def to_python(self, value):
150:     def get_db_prep_save(self, value, *args, *kwargs):
```

/src/protoLib/models.py :

Définit les concepts additionnels pour compléter la structure de Django pour les besoins des applications génériques, dont, la sécurité.

/src/protoLib/models.py:

```
  11: class TeamHierarchy(models.Model):
  21:     def fullPath(self):
  25:     def treeHierarchy(self):
  35:     def save(self, *args, *kwargs):

  50: class UserProfile(models.Model):
  64: def user_post_save(sender, instance, created, *kwargs):

  74: class UserShare(models.Model):
  85: class ProtoModel(models.Model):
 104: class EntityMap(models.Model):
 140: class FieldMap(models.Model):
 158: class ProtoDefinition(models.Model):
 191: class CustomDefinition(ProtoModel):

 216: def getDjangoModel(modelName):
 238: def getNodeHierarchy(record, parentField,  codeField, pathFunction ):

 249: class DiscreteValue(models.Model):
 275: class Languaje(models.Model):
 295: class PtFunction(models.Model):
```

/src/protoLib/protoActionEdit.py :

Liste de classes et méthodes

/src/protoLib/protoActionEdit.py:

```
  23: def protoCreate(request):
  28: def protoUpdate(request):
  32: def protoDelete(request):
  36: def _protoEdit(request, myAction):
 193: def setSecurityInfo(rec, data, userProfile, insAction ):
 212: def setProtoData(rec, data, key, value ):
 218: def setRegister(model,  rec, key,  data  ):
```

/src/protoLib/protoActionList.py :

Liste de classes et méthodes

```
/src/protoLib/protoActionList.py:
   27: def protoList(request):
   94: def Q2Dict ( protoMeta, dataRows, fakeId ):
  231: def getRowById(myModelName, myId):
  241: def getUserNodes(pUser, viewEntity):
  248: def getQSet( protoMeta, protoFilter, baseFilter , sort , pUser ):
  321: def getUnicodeFields(model):
  332: def evalueFuncion(fName, dataReg):
  343: def PrepareMeta2Load(protoMeta):
```

/src/protoLib/protoActionRep.py :

Liste de classes et méthodes

```
/src/protoLib/protoActionRep.py:
   24: def sheetConfigRep(request):
   82: def getSheetConf(protoMeta , sheetName):
  108: def getReportBase(viewCode):
  133: class SheetReportFactory(object):
  137:     def __init__(self):
  143:     def getReport(self , Qs, templateBefore, templateERow, templateAfter, protoMeta, s
  210: def getProperties(fields, template):
  226: def getDetailConf(protoMeta, detailName):
  244: def getReport(props, template, row ):
  257: def getLineCsv(line):
  264: def protoCsv(request):
```

/src/protoLib/protoActions.py :

Liste de classes et méthodes

```
/src/protoLib/protoActions.py:
   11: def protoExecuteAction(request):
```

/src/protoLib/protoAuth.py :

Liste de classes et méthodes

```
/src/protoLib/protoAuth.py:
    7: def  getUserProfile(pUser, action, actionInfo ):

   62: def getModelPermissions(pUser, model , perm = None):
   70:     def getIndPermission(perm ):
   92: def activityLog(action, user, option, info ):

  137: class Task(models.Model):
```

/src/protoLib/protoField.py :

Liste de classes et méthodes

```
/src/protoLib/protoField.py:
   25: def setFieldDict(protoFields , field):
  152: def setFieldProperty(pField, pProperty, pDefault, field, fProperty, fpDefault):
  161: def isAdmField(fName ):
```

/src/protoLib/protoGetDetails.py :

Liste de classes et méthodes

```
/src/protoLib/protoGetDetails.py:
   13: def protoGetDetailsTree(request):
   57: def addDetailToList( detailList , detail, detailPath ):
```

/src/protoLib/protoGetPci.py :

Liste de classes et méthodes

```
/src/protoLib/protoGetPci.py:
   28: def protoGetPCI(request):
  163: def createProtoMeta(model, grid, viewEntity , viewCode):
  246: def protoSaveProtoObj(request):
  335: def protoGetFieldTree(request):
  395: def addFiedToList( fieldList , field, fieldBase ):
  469: def isFieldDefined(pFields , fName):
```

/src/protoLib/protoGrid.py :

Liste de classes et méthodes

```
/src/protoLib/protoGrid.py:
   40: class ProtoGridFactory(object):

   11: def getProtoAdmin(model):
   44:     def __init__(self, model, viewCode, model_admin, protoMeta ):
  145:     def getFieldSets(self):
  261:     def get_details(self):
  274: def getModelDetails(model):
  323: def setDefaultField (fdict, model, viewCode):
  339: def getBaseModelName(viewCode ):
  351: def getFieldsInSet(self, prItems, formFields):
  366: def verifyField(self, fName):
```

/src/protoLib/protoLogin.py :

Liste de classes et méthodes

```
/src/protoLib/protoLogin.py:
   11: def protoGetUserRights(request):
```

/src/protoLib/protoMenu.py :

Liste de classes et méthodes

```
/src/protoLib/protoMenu.py:
   26: def protoGetMenuData(request):
   51:     def getMenuItem(protoAdmin, model, menuNode):
```

/src/protoLib/protoQbe.py :

Liste de classes et méthodes

```
/src/protoLib/protoQbe.py:
   20: def addFilter(Qs, sFilter):
   48: def construct_search(field_name):
   61: def getSearcheableFields( model ):
   77: def getQbeStmt(fieldName , sQBE, sType ):
  191: def doGenericFuntion(sQBE):
  222: def addQbeFilter(protoFilter, model, Qs, JsonField):
  255: def addQbeFilterStmt(sFilter, model, JsonField):
  288: def getTextSearch(sFilter, model , protoMeta):
```

/src/protoLib/usrDefProps.py :

Liste de classes et méthodes

```
/src/protoLib/usrDefProps.py:
   11: def verifyUdpDefinition(pUDP):
   49: def saveUDP(regBase, data, cUDP ):
  100: def readUdps(rowdict, regBase , cUDP, udpList, udpTypes):
```

/src/protoLib/utilsBase.py :

Liste de classes et méthodes

```
/src/protoLib/utilsBase.py:
   20:     def default(self, obj):
   30: def verifyList(obj):
```

```
 43: def verifyStr(vrBase , vrDefault):
 48: def parseEmailAddress(fullemail, delimitorLeft = '<', delimitorRight = '>'):
 74: def guessNextPath(dst, slugify = True, idx = 0, checkExists = True):
 96: def unique_id(more = ''):
107: def reduceDict(inDict, keep_keys):
116: def dict2tuple(indict):
124: def list2dict(alist , key):
140: def CleanFilePath(inFileName):
148: def CheckPathSecurity(testPath, rootPath):
152: def ReadFile(inFile, mode='r'):
162: def WriteFile(inFile, contents):
167: def PathToList(inPath, template_type="", showExt = True):
179: def strip_html(inHtml):
192: def strip_accents(inStr):
207: def strip_euro(inStr):
215: def DateFormatConverter(to_extjs = None, to_python = None):
259:     def __init__(self, name):
263: def getReadableError(e):
271: def strNotNull( sValue, sDefault):
279: def copyProps (objBase, objNew):
288: def explode(s):
306: def findBrackets(aString):
321: #def balanced_braces(args):
347: #def mbrack(s):
406: def slugify(text, delim= '-'):
```

/src/protoLib/utilsConvert.py :

Liste de classes et méthodes

```
/src/protoLib/utilsConvert.py:
  8: def getTypedValue (sAux , sType):
 34: def isNumeric(s):
 43: def toInteger(s , iDefault = None):
 54: def toFloat(s , iDefault = None):
 65: def toDecimal(s , iDefault = None):
 77: def toBoolean(s):
 91: def toDate(sVal, iDefault = None):
 99: def toTime(sVal, iDefault = None):
107: def toDateTime(sVal, iDefault = None):
138: def toDate__(sVal):
165: def isinteger(astring):
```

/src/protoLib/utilsDb.py :

Liste de classes et méthodes

```
/src/protoLib/utilsDb.py:
    3: def setDefaults2Obj(pObj, defaults, exclude = []):
   15: def update_or_create(myModel , *kwargs):
```

/src/protoLib/utilsWeb.py :

Liste de classes et méthodes

```
/src/protoLib/utilsWeb.py:
   17: def doReturn(jsonDict):
   22: def JsonResponse(contents, status=200):
   26: def JsonSuccess(params = {}):
   31: def JsonError(error = ''):
   35: def set_cookie(response, key, value, days_expire = 7):
   47: def get_cookie(request, key):
   52: def DownloadLocalFile(InFile):
   62: def JSONserialise(obj):
   70: def my_send_mail(subject, txt, sender, to=[], files=[], charset='UTF-8'):
```

/src/protoLib/actions/__init__.py :

Liste de classes et méthodes

```
/src/protoLib/actions/__init__.py:
    7: def doFindReplace(modeladmin, request, queryset, parameters):
```

/src/protoLib/actions/findReplace.py :

Liste de classes et méthodes

```
/src/protoLib/actions/findReplace.py:
    6: def actionFindReplace(request,  queryset , parameters):
```

/src/protoLib/management/__init__.py :

Liste de classes et méthodes

```
/src/protoLib/management/__init__.py:
    5: def addProtoPermissions(sender, *kwargs):
   10:     def addEntityPermission(label, title):
```

/src/protoLib/meta/csvEport.py :

Liste de classes et méthodes

```
/src/protoLib/meta/csvEport.py:
    4: def excelview(request):
   17:     def __init__(self, data, output_name='excel_data', headers=None,
```

/src/protoLib/utils/authPlugins.py :

Liste de classes et méthodes

```
/src/protoLib/utils/authPlugins.py:
    2:     def __init__(cls, name, bases, attrs):
   28: def is_valid_password(password):
   39: def get_password_errors(password):
   53:     def validate(self, password):
   59:     def validate(self, password):
```

/src/protoLib/utils/xml2json.py :

Liste de classes et méthodes

```
/src/protoLib/utils/xml2json.py:
   40: def elem_to_internal(elem,strip=1):
   81: def internal_to_elem(pfsh, factory=ET.Element):
  122: def elem2json(elem, strip=1):
  131: def json2elem(json, factory=ET.Element):
  143: def xml2json(xmlstring, strip=1):
  151: def json2xml(json, factory=ET.Element):
  164: def main():
```

/src/protoLib/usrDefProps.py :

Liste de classes et méthodes

```
/src/protoLib/usrDefProps.py:
    8: class cAux: pass
```

/src/protoLib/utilsBase.py :

Liste de classes et méthodes

```
/src/protoLib/utilsBase.py:
   19: class JSONEncoder(json.JSONEncoder):
  258: class VirtualField(object):
```

/src/protoLib/utilsDb.py :

Liste de classes et méthodes

```
/src/protoLib/utilsDb.py:
   20:      class PersonManager(models.Manager):
   23:      class Person(models.Model):
```

/src/protoLib/admin/adminOrgTree.py :

Liste de classes et méthodes

```
/src/protoLib/admin/adminOrgTree.py:
    3: class orgTreeAdmin(django.contrib.admin.ModelAdmin):
```

/src/protoLib/admin/adminProtoDef.py :

Liste de classes et méthodes

```
/src/protoLib/admin/adminProtoDef.py:
    5: class protoDefinitionAdmin(django.contrib.admin.ModelAdmin):
```

/src/protoLib/admin/adminUserProf.py :

Liste de classes et méthodes

```
/src/protoLib/admin/adminUserProf.py:
    3: class usrProfileAdmin(django.contrib.admin.ModelAdmin):
```

/src/protoLib/meta/csvEport.py :

Liste de classes et méthodes

```
/src/protoLib/meta/csvEport.py:
   15: class ExcelResponse(HttpResponse):
```

/src/protoLib/utils/authPlugins.py :

Liste de classes et méthodes

```
/src/protoLib/utils/authPlugins.py:
    1: class PluginMount(type):
    6:             # class shouldn't be registered as a plugin. Instead, it sets up a
   15: class PasswordValidator(object):
   17:     Plugins extending this class will be used to validate passwords.
   52: class MinimumLength(PasswordValidator):
   58: class SpecialCharacters(PasswordValidator):
```

/src/protoLib/utils/xml2json.py :

Liste de classes et méthodes

```
/src/protoLib/utils/xml2json.py:
   87:     Element class as the factory parameter.
  136:     default; if you want to use something else, pass the Element class
  156:     default; if you want to use something else, pass the Element class
```

"/src/prototype"

Voici un liste des services génériques exécutés lors de l'utilisation de l'application :

/src/prototype/models.py :

Liste de classes et méthodes

```
/src/prototype/models.py:
   51: class Project(ProtoModel):
   80: class Model(ProtoModel):
  116: class Entity(ProtoModel):
  248: class ForeignEntity(ProtoModel):
  274: class PropertyBase(ProtoModel):
  302: class Property(PropertyBase):
  364: class Relationship(Property):
  405: class PropertyProject(PropertyBase):
  478: class PropertyEquivalence(ProtoModel):
  536: class Prototype(ProtoModel):
  564: class ProtoTable(ProtoModel):
  596: class Diagram(ProtoModel):
  624: class DiagramEntity(ProtoModel):
  652: class Service(ProtoModel):
  680:     class Meta:
  687: class ServiceRef(ProtoModel):
  708:     class Meta:
```

/src/prototype/models.py :

Liste de classes et méthodes

```
/src/prototype/models.py:
  339:     def save(self, *args, *kwargs):
  389:     def save(self, *args, *kwargs):
  429:     def save(self, *args, *kwargs):
  468: def propModel_post_delete(sender, instance, *kwargs):
  503:     def delete(self, *args, *kwargs):
  515: def propEquivalence_post_save(sender, instance, created, *kwargs):
```

/src/prototype/admin.py :

Liste de classes et méthodes

```
/src/prototype/admin.py:
   17: class MyModelAdmin(admin.ModelAdmin):
   28: class MyEntityAdmin(admin.ModelAdmin):
   37: class MyPropertyProjectAdmin(admin.ModelAdmin):
   45: class MyProjectAdmin(admin.ModelAdmin):
```

/src/prototype/protoRules.py :

Liste de classes et méthodes

```
/src/prototype/protoRules.py:
   44: def updatePropInfo(myBase, propBase, modelBase, inherit ):
   97: def twoWayPropEquivalence(propEquiv, modelBase, deleted):
  147: def updPropertyProject(Property):
```

/src/prototype/actions/__init__.py :

Liste de classes et méthodes

```
/src/prototype/actions/__init__.py:
   11: def doModelPrototype(modeladmin, request, queryset, parameters):
   32: def doEntityPrototype(modeladmin, request, queryset, parameters):
   47: def doPropertyProjectJoin(modeladmin, request, queryset, parameters):
   61: def doPropertyProjectPurge(modeladmin, request, queryset, parameters):
   76: def doModelGraph(modeladmin, request, queryset, parameters):
  118: def doAutoForeingEntity(modeladmin, request, queryset, parameters):
  134: def doImportSchema(modeladmin, request, queryset, parameters):
  136:     funcion para Importar la def de una Db (basado en inspectDb)
  163: def doEntityChangeModel(modeladmin, request, queryset, parameters):
```

/src/prototype/actions/entityActions.py :

Liste de classes et méthodes

```
/src/prototype/actions/entityActions.py:
    8: def doEttyChangeModel(request,  queryset , parameters):
```

/src/prototype/actions/graphModel.py :

Liste de classes et méthodes

```
/src/prototype/actions/graphModel.py:
   13: def generateDotModels(queryset):
```

/src/prototype/actions/modelActions.py :

Liste de classes et méthodes

```
/src/prototype/actions/modelActions.py:
    6: def actionAutoForeingEntity(queryset ):
```

/src/prototype/actions/projectActions.py :

Liste de classes et méthodes

```
/src/prototype/actions/projectActions.py:
   17: #def createNewModel(modeladmin, request, queryset):
   46: #def getModel(objDomain, modelCode ):
```

/src/prototype/actions/propModelJoin.py :

Liste de classes et méthodes

```
/src/prototype/actions/propModelJoin.py:
    6: def doPropModelJoin(queryset):
```

/src/prototype/actions/reverseDb.py :

Liste de classes et méthodes

```
/src/prototype/actions/reverseDb.py:
   28: def getDbSchemaDef(dProject , request ):
  197: def saveProperty(dEntity, pProperty, defValues, userProfile, prpName, seq
):
  218: def saveRelation(dProject, dEntity, dModel, pProperty,  defValues, userProfile, prpName, se
):
  260: def get_field_type(connection, table_name, row):
```

/src/prototype/actions/viewDefinition.py :

Liste de classes et méthodes

```
/src/prototype/actions/viewDefinition.py:
   15: def getViewDefinition(pEntity, viewTitle ):
   78: def GetDetailsConfigTreeById(protoEntityId):
   92: def GetDetailsConfigTree(pEntity):
  118: def getViewCode(pEntity, viewTitle = None):
  124: def property2Field(fName, propDict, infoField = False, fBase = ''):
  154: def getFkId(fName, infoField = False, fBase = ''):
```

```
177: def GetProtoFieldsTree( protoEntityId):
192: def addProtoFiedToList(fieldList , pEntity , fieldBase, zoomName ):
243: def getEntities(queryset , request, viewTitle ):
257: def createView(pEntity, viewTitle, userProfile):
```

/src/prototype/actions/viewTemplate.py :

Liste de classes et méthodes

```
/src/prototype/actions/viewTemplate.py:
    8: def baseDefinition(pEntity , entityName, viewTitle ):
```

"/xml2db"

Voici un liste des services génériques exécutés lors de l'utilisation de l'application :

/xml2db/exportDj.py :

Liste de classes et méthodes

```
/xml2db/exportDj.py:
    7:      def __init__(self):
   12:      def ExportDjangoModel(self):
   17:      def ExportDjangoModel_body(self):
   21:      def runJob(self):
   48:      def generateProject(self, iw):
   58:      def iniFileModel(self, iw):
   70:      def iniFileAdmin(self, projectName):
   86:      def generateDataModel(self, iw, model, idRef):
  106:      def generateTable(self, iw, model, table):
  146:          iw.println("def __unicode__(self):")
  204:      def generateAdminReg(self, wTable):
  210:      def generateColumn(self, iw, wTable, col):
```

/xml2db/exportDj.py :

Liste de classes et méthodes

```
/xml2db/exportDj.py:
    5: class ExportDjangoModel(object):
  107:          pattern = "class {0}(models.Model):"
  196:              iw.println("class Meta: ")
```

/xml2db/importDict.py :

Liste de classes et méthodes

```
/xml2db/importDict.py:
   25: def toInteger(s , iDefault = None):
   35: def toBoolean(s):
   43:     def __init__(self):
   67:     def loadFile(self, filename):
   96:     def getFilename(self):
  100:     def getContentFile(self):
  116:     def __write(self):
  313:     def writeDatabase(self):
  321:     def saveModelUdps(self, udps, dModel):
  332:     def savePrpUdps(self, udps, dPrp):
  344: def getModelRef(dProject, modelName ):
  349: def getPrpRef(dProject , propName ):
  354: def getConceptRef(dModel , cName ):
```

/xml2db/importDict.py :

Liste de classes et méthodes

```
/xml2db/importDict.py:
   42: class importDict():
```

/xml2db/main.py :

Liste de classes et méthodes

```
/xml2db/main.py:
   16: def main():
```

/xml2db/systemGui.py :

Liste de classes et méthodes

```
/xml2db/systemGui.py:
    7:     def __init__(self, systemCore):
   26:     def __load(self):
   42:     def __convertToDb(self):
   56:     def __exportXML(self):
   60:     def __setInitialMenu(self):
```

/xml2db/systemGui.py :

Liste de classes et méthodes

/xml2db/testSystemApp.py:
```
  6: class systemGui(QtGui.QMainWindow):
 50: class testSystemApp(unittest.TestCase):
```

/xml2db/testSystemApp.py :

Liste de classes et méthodes

/xml2db/testSystemApp.py:
```
 53:     def setUp(self):
 56:     def testLoadFilenameFichierValide(self):
 71:     def testLoadFilenameFichierVide(self):
 79:     def testLoadFilenameFichierNonValide(self):
 87:     def testWriteDatabaseConnexion(self):
101:     def testVerifyWriteDatabase(self):
```

Bibliographie

[1] Code-sharing site github turns five and hits 3.5 million users, 6 million repositories. 2.6

[2] Coding style | django documentation | django. 5.2.2

[3] Définitions : méthodologie - dictionnaire de français larousse. 8

[4] The DOT language. A, 9

[5] Forever for now - UML diagrams using graphviz dot. 9

[6] Introduction to the togaf ADM. (document), 2.5

[7] Ladot : LaTeX in your graphviz dot files. 9

[8] ONTOLOGIQUE : Définition de ONTOLOGIQUE. 2.4.2

[9] Open ModelSphere - outils de modélisation logiciel libre GPL. 4

[10] PEP 8 – style guide for python code. 5.2.2

[11] Web services @ W3C. 5.6

[12] What is scrum ? | scrum.org - the home of scrum. (document), 2.5.3, 2.8

[13] NATO software engineering conference 1968, 1968. 0.1

[14] Facebook re-write takes PHP to an enterprise past • the register, 2010. 2.6

[15] CamelCase, August 2013. Page Version ID : 96109883. 5.2.2

[16] QQOQCCP, August 2013. 16

[17] David Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. In *« UML» 2002—The Unified Modeling Language*, page 243–258. Springer, 2002. 3.3.1

[18] Atia M. Albhbah and Mick J. Ridley. Using RuleML and database metadata for automatic generation of web forms. In *Intelligent Systems Design and Applications (ISDA), 2010 10th International Conference on*, page 790–794, 2010. 00005. 1.1, 1.2.3, 1.3

[19] Atia M Albhbah and Mick J Ridley. An extended rule framework for web forms : adding to metadata with custom rules to control appearance. 2011. 00000. 1.1, 1.2.3, 1.3

[20] Atia M. Albhbah and Mick J. Ridley. A rule framework for automatic generation of web forms'. In *Proceedings of the 4th IEEE International Conference on Computer Science and Information Technology (IEEE ICCSIT 2011), Chengdu, China*, 2012. 00001. 0.3.1, 0.3.2, 1.1, 1.2.3, 1.3

[21] Scott Ambler. *Agile modeling : effective practices for extreme programming and the unified process*. Wiley. com, 2002. 2.6.1

[22] Scott Ambler. Agile model driven development (AMDD) : the key to scaling agile software development, 2012. 0.3.2

[23] Scott Ambler. The object-relational impedance mismatch, 2012. 2.1, 2

[24] Stephen J. Andriole. Fast, cheap requirements prototype, or else ! *Software, IEEE*, 11(2) :85–87, 1994. 0.3.1

[25] I. Antovic ?, S. Vlajic ?, M. Milic ?, D. Savic ?, and V. Stanojevic ? Model and software tool for automatic generation of user interface based on use case and data model. *IET Software*, 6(6) :559, 2012. 00000. 0.2, 0.3.1, 0.3.2, 1.1, 1.2.1, 1.2.3, 1.3

[26] Colin Atkinson, Christian Bunse, and Jürgen Wüst. Driving component-based software development through quality modelling. In *Component-Based Software Quality*, page 207–224. Springer, 2003. 2.3.6

[27] Colin Atkinson and Thomas Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36–41, 2003. (document), 0.3, 0.3.2, 3.3.1, 3.2, 3.3.1, 3.3, 3.3.1

[28] Abel Avram and Floyd Marinescu. Domain-driven design quickly : [a summary of eric evans' domain-driven design]. 2006. 0.3.2, 4.1

[29] Abel Avram and Floyd Marinescu. *Domain-Driven Design (vite fait)*. C4Media, [S.l.], 2006. 2.6.1, 3

[30] Thomas Beale. Archetypes : Constraint-based domain models for future-proof information systems. In *OOPSLA 2002 workshop on behavioural semantics*, page 1–18, 2002. 21

[31] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, and Ron Jeffries. Manifesto for agile software development. 2001. 0.3.1, 2.5.3

[32] Mariano Belaunde, Cory Casanave, Desmond DSouza, Keith Duddy, William El Kaim, Alan Kennedy, William Frank, David Frankel, Randall Hauch, Stan Hendryx, and OMG. MDA guide version 1.0.1. 2003. 2.3.7, 3.2.1

[33] Herbert D Benington. Production of large computer programs. *Annals of the History of Computing*, 5(4) :350–361, 1956. (document), 2.3.2, 2.5, 2.5.1, 2.6

[34] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5) :28–37, 2001. 7.7

[35] Larry Bernstein. Foreword : Importance of software prototyping. *Journal of Systems Integration*, 6(1) :9–14, 1996. 0.3, 0.3.1, 0.3.1

[36] V. Berzins, M. Shing, N. Nada, and C. Eagle. Software evolution approach for the development of command and control systems. Technical report, DTIC Document, 2000. 0.3.1

[37] Jean Bézivin. MDA : from hype to hope, and reality. *Invited talk at UML*, 2003. 0.3.2, 2.3.7, 3.2.1

[38] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2) :171–188, 2005. 3.1

[39] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, page 273–280, 2001. 0.3, 0.3.2

[40] Barry Boehm. Get ready for agile methods, with care. *Computer*, 35(1) :64–69, 2002. 2.6.1

[41] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5) :61–72, 1988. (document), 0.2, 0.2, 2.5.2

[42] Gregory W Bond. Software as art. *Communications of the ACM*, 48(8) :118–124, 2005. 3.2

[43] Grady Booch, Douglas L Bryan, and Charles G Petersen. *Software engineering with Ada*. Addison-Wesley Professional, 1994. 0.1

[44] Frederick P. Brooks. No silver bullet : Essence and accidents of software engineering. *IEEE computer*, 20(4) :10–19, 1987. 0.2, 2, 5.1.2

[45] Mahil Carr and June Verner. Prototyping and software development approaches. Technical report, Working Paper, downloaded from : http ://www. is. cityu. edu. hk/Research/WorkingPapers/paper/9704. pdf, 1997. 0.3.1, 1

[46] Peter Checkland. Soft systems methodology : a thirty year retrospective, 1981. 7

[47] Peter Checkland. SOFT SYSTEMS METHODOLOGY, 1990. 2.5.3, 4

[48] Liguang Chen. *An Empirical investigation into management and control of software prototyping.* PhD thesis, Bournemouth University, 1997. 0.1

[49] Edgar F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4) :397–434, 1979. 4.1

[50] Edgar F Codd. *The relational model for database management : version 2.* Addison-Wesley Longman Publishing Co., Inc., 1990. 4.1

[51] David Cohen, Mikael Lindvall, and Patricia Costa. An introduction to agile methods. In *Advances in Computers*, volume 62, pages 1–66. Elsevier, 2004. 2.6.1

[52] Hugo Corbucci and Alfredo Goldman. Open source and agile : Two worlds that should have a closer interaction. 2008. 2.5.4

[53] J. Den Haan. Model driven development : Code generation or model interpretation ?, 2010. 1.2.2

[54] J. Den Haan, A. Albani, and J. L. G. Dietz. *An Enterprise Ontology based approach to Model-Driven Engineering.* PhD thesis, October 2009. 0.3.2, 2.3.6, 3.1

[55] Google Dev. Google and the open source developer, 2013. 2.6

[56] B. A. Devlin and P. T. Murphy. An architecture for a business and information system. *IBM Systems Journal*, 27(1) :60–80, 1988. 2.3.2

[57] Nicolas Dieudonne. De l'integration des applications... a l'integration de l'organisation : l'urbanisation du systeme d'information. 2006. 2.4.3

[58] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10) :859–866, 1972. 0.1

[59] Edsger W Dijkstra. How do we tell truths that might hurt ? In *Selected Writings on Computing : A Personal Perspective*, page 129–131. Springer, 1975. (document)

[60] Edsger W Dijkstra and WH Feijen. *A method of programming.* Addison-Wesley Longman Publishing Co., Inc., 1988. 0.2, 2.3.3

[61] Michel Diviné and Hubert Tardieu. *Parlez-vous Merise ?* Eyrolles, 1989. 2.3.2, 10

[62] Merlin Dorfman and Richard H Thayer. *Software requirements engineering.* IEEE Computer Society Press, 2000. (document), 2.10, 2.7

[63] Amnon H. Eden. A theory of object-oriented design. *Information Systems Frontiers*, 4(4) :379–391, 2002. 0.1

[64] Mohammed M Elsheh and Mick J Ridley. Using database metadata and its semantics to generate automatic and dynamic web entry forms in. In *Proceedings of the World Congress on Engineering and Computer Science 2007 WCECS 2007*, 2007. 1.1, 1.2.3

[65] Eric Evans. *Domain-driven Design : Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. 0.3, 0.3.2, 2.4, 2.7, 3.1, 3.2, 3.2.2, 2, 3.2.2, 3.4

[66] Liliana Favre and Liliana Martinez. Formalizing MDA components. In *Reuse of Off-the-Shelf Components*, page 326–339. Springer, 2006. 2.3.6

[67] Joseph Feller and Brian Fitzgerald. A framework analysis of the open source software development paradigm. In *Proceedings of the twenty first international conference on Information systems*, page 58–69, 2000. 0.1

[68] James D Foley et al. History, results, and bibliography of the user interface design environment (UIDE), an early model-based system for user interface design and implementation. In *Interactive Systems : Design, Specification, and Verification*, page 3–14. Springer, 1995. 2.3.7

[69] Andrew Forward, O. Badreddin, and T. C. Lethbridge. Perceptions of software modeling : a survey of software practitioners. In *5th workshop from code centric to model centric : evaluating the effectiveness of MDD (C2M : EEMDD)*, 2010. 0.3.2

[70] Andrew Forward, Omar Badreddin, Timothy C. Lethbridge, and Julian Solano. Model-driven rapid prototyping with umple. *Software : Practice and Experience*, 42(7) :781–797, 2011. 00000. 0.3.2, 1.1, 1.2.3, 1.3

[71] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development : a survey of software professionals. In *Proceedings of the 2008 international workshop on Models in software engineering*, page 27–32, 2008. 0.3, 0.3.2

[72] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003. 2.4, 2.6.1, 5.6

[73] Martin Fowler. The new methodology, 2005. 2.5, 2.5.1, 2.5.3, 2.5.3, 2.5.4, 2.6.1, 2.7

[74] Martin Fowler. *Domain-specific languages*. Addison-Wesley, Upper Saddle River, NJ, 2011. 1.2.2, 5

[75] Martin Fowler. Refactoring home, 2012. 2.6.1

[76] Alison A Frost and Michael J Campo. Advancing defect containment to quantitative defect management. 2007. 00015. (document), 0.2, 0.1

[77] Lidia Fuentes and Pablo Sánchez. Towards executable aspect-oriented UML models. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, page 28–34, 2007. 2.6.1

[78] Alfonso Fuggetta. Open source software– an evaluation. *Journal of Systems and Software*, 66(1) :77–90, April 2003. 2.6

[79] E Gamma, John Vlissides, R Helm, and R Johnson. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass., 1995. A

[80] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. *Retrieved June*, 13, 2010. 9

[81] Jesse James Garrett. *Ajax : A new approach to web applications*. 2005. 5.6

[82] Gilb. Value"to"Stakeholders"–" not"working"code"to"customers, 2010. 2.5.3

[83] Tom Gilb. Evolutionary delivery versus the waterfall model. *ACM SIGSOFT Software Engineering Notes*, 10(3) :49–61, 1985. 2.5.1

[84] Ambrose Goikoetxea. *Enterprise architectures and digital administration : Planning, design and assessment*. World Scientific Publishing Company, 2007. 2.3.2, 2.4.2

[85] Open Group. *TOGAF Version 9*. Van Haren Pub., Zaltbommel [Netherlands], 2009. 2.4.3

[86] Peter Hantos. From spiral to anchored processes : A wild ride in lifecycle architecting. In *Proceedings, USC-SEI Spiral Experience Workshop*, 2000. 5.1.2

[87] Wilhelm Hasselbring. Component-based software engineering. *Handbook of software engineering and knowledge engineering*, 2 :289–305, 2002. 2.3.6

[88] Ellis Horowitz and Barry W Boehm. *Practical strategies for developing large software systems*. Addison-Wesley, 1975. 0.2, 0.3, 0.3.1, 2.7

[89] Ayman Hourieh and Susmita Basu. *Learning website development with Django a beginner's tutorial to building web applications, quickly and cleanly, with the Django application framework*. Packt Pub., Birmingham, U.K., 2008. 1.2.3

[90] Zachman International. The zachman framework™ : The official concise definition, 2013. (document), 2.4.2, 2.1

[91] Pontus Johnson, Robert Lagerstrom, Per Norman, and Marten Simonsson. Extended influence diagrams for enterprise architecture analysis. In *Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International*, page 3–12, 2006. 2.4.3

[92] Rajive Joshi. Data-oriented architecture : A loosely-coupled real-time SOA. *Real-Time Innovations, Inc, CA, Tech. Rep*, 2007. 2.3.4, 2.3.5, 5.6

[93] K Kendall and J Kendall. Systems analysis and design. 2013. (document), 2.9, 2.6.1

[94] Jinwoo Kim, F Javier Lerch, and Herbert A Simon. Internal representation and rule development in object-oriented design. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(4) :357–390, 1995. 2.3.5

[95] Donald E. Knuth. Letter to the patent office from donald knuth, 1994. 18

[96] Donald E. Knuth and Andrew Binstock. Interview with donald knuth. April 2008. 2.6

[97] Karl E. Kurbel. *The Making of Information Systems : Software Engineering and Management in a Globalized World*. Springer, April 2008. 2.4.3

[98] Strahinja Lazetic, Dusan Savic, Sinisa Vlajic, and Sasa Lazarevic. A generator of MVC-based web applications. *World of Computer Science and Information Technology Journal (WCSIT)*, 2(4) :147–156, 2012. 1.2.3

[99] Jean-L Le Moigne. Conception de la complexité et complexité de la conception. *Revue Internationale de Systémique*, 4(2) :295–318, 1990. 2.3.1, 6, 3.3

[100] Jean-Louis Le Moigne. *Les systèmes d'information dans les organisations*. Presses universitaires de France, 1973. 2.3.4

[101] Jean-Louis Le Moigne. *Qu'est-ce qu'un modèle*. Université d'Aix-Marseille III, Faculté d'économie appliquée, 1987. 1, 2.3.1

[102] Jean-Louis Le Moigne. *La modélisation des systèmes complexes*, volume 2. Dunod Paris, 1990. 2.3.1

[103] Jean-Louis Le Moigne. LES FORMALISMES DE LA MODELISATION SYSTEMIQUE. 1992. 2.3.4, 3

[104] Jean-Louis Le Moigne. *La théorie du système général : théorie de la modélisation*. Presses Universitaires de France-PUF, 1994. 2.2, 2.3.1, 2.3.4, 2.3.5

[105] Jean Louis Le Moigne. Nous pouvons maintenant comprendre ce qu'est un processus de conception, 2013. 2.2, 2.3.1

[106] Jean-Louis Le Moigne and Edgar Morin. *L'intelligence de la complexité*. Harmattan, 1999. 2.5.4

[107] Jean Louis Le Moigne and Daniel Pascot. *Les Processus collectifs de mémorisation (mémoire et organisation) : actes du Colloque d'Aix-en-Provence GRASCE–Faculté d'Économie Appliquée–(Juin 1979)*. Librairie de l'Université, 1979. 2.3.4

[108] Richard Lemesle. Meta-modeling and modularity : Comparison between MOF, CDIF and sNets formalisms. In *OOPSLA'98 Workshop# 25 : Model Engineering, Methods and Tools Integration with CDIF*, 1998. 3.3.1

[109] Yuan Li, Xiu Wu Liao, and Hong Zhen Lei. A knowledge management system for ERP implementation. *Systems Research and Behavioral Science*, 23(2) :157–168, April 2006. 0.1

[110] Youn-Kyung Lim, Erik Stolterman, and Josh Tenenberg. The anatomy of prototypes : Prototypes as filters, prototypes as manifestations of design ideas. *ACM Transactions on Computer-Human Interaction*, 15(2) :1–27, July 2008. 0.3.1, 0.3.1

[111] Liza Daly. *Next-generation web frameworks in Python*. O'Reilly, Sebastopol, Calif., 2007. 4.3.3

[112] Christophe Longépé. Le projet d'urbanisation du SI. *Collection Informatique et Entreprise, Dunod*, 2001. 2.4.3

[113] Stephen J Mellor and Marc J Balcer. *Executable UML : a foundation for model-driven architecture*. Addison-Wesley Professional, 2002. 2.6.1

[114] Mohamed A Mgheder and Mick J Ridley. Automatic generation of web user interfaces in PHP using database metadata. In *Internet and Web Applications and Services, 2008. ICIW'08. Third International Conference on*, page 426–430, 2008. 00007. 1.1, 1.2.3, 1.3

[115] M. Missikoff and R. Pizzicannella. A visual approach to object-oriented analysis based on abstract diagrams. *ACM SIGCHI Bulletin*, 28(3) :56–64, 1996. 2.3.5, 13

[116] OMG Mof. XMI mapping specification, v2. 41. *OMG Document*, page 05–09, 2013. 4.2.2

[117] Eben Moglen. Freeing the mind : Free software and the death of proprietary culture. *Me. L. Rev.*, 56 :1, 2004. 2.6

[118] A. Momoh, R. Roy, and E. Shehab. Challenges in enterprise resource planning implementation : state-of-the-art. *Business Process Management Journal*, 16(4) :537–565, 2010. 0.1

[119] John Mylopoulos. Metamodeling, 2004. 0.3.2, 3.3

[120] Sridhar Nerur, RadhaKanta Mahapatra, and George Mangalaraj. Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5) :72–78, 2005. (document), 2.2, 2.5.3

[121] OMG. Object management group. 2.3.5, 12

[122] OMG. *OMG Meta Object Facility (MOF) Specification v1. 4.* OMG Document formal/02-04-03 [Online]. Available : http ://www. omg. org/cgibin/apps/doc, 2002. 3.3.1

[123] OMG. OMG official MDA webpage, 2003. 0.3.2, 3.2.1

[124] OMG. *Unified Modeling Language Specification v. 1.5.* March, 2003. 3.3.1

[125] OMG. Semantics of a foundational subset for executable UML models (fUML), v1.1 RTF beta. *Object Management Group pct/07-08-04*, 2012. 0.3.2, 2.6.1, 3.1

[126] OMG. Semantics of a foundational subset for executable UML models (fUML), 2013. 22

[127] Burt Parker. Introducing ANSI-X3. 138-1988 : a standard for information resource dictionary system (IRDS). In *Assessment of Quality Software Development Tools, 1992., Proceedings of the Second Symposium on*, page 90–99, 1992. (document), 3.3.1, 3.1

[128] D Pascot. *La methode Datarun, Management Information Systems Department, Faculty of Business Administration, Laval University, Quebec, Canada.* 1999. 1.2.1, 2.7

[129] Daniel Pascot. DATARUN concepts, 1996. 1.2.1, 2.3.4, 2.7, 3.1

[130] Daniel Pascot. La réalisation des prototypes d'applications de gestion suivant datarun, 2001. 0.3.1, 0.3.1, 1.2.1

[131] Daniel Pascot. L'effet de la stratégie de logiciel metier sur le capital humain organisationnel accessible, April 2008. 19

[132] Daniel Pascot, Faouzi Bouslama, and Sehl Mellouli. Architecturing large integrated complex information systems : an application to healthcare. *Knowledge and Information Systems*, 27(1) :115–140, March 2010. 2.7

[133] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10) :14–17, 2005. 5.6

[134] Carla Marques Pereira and Pedro Sousa. A method to define an enterprise architecture using the zachman framework. In *Proceedings of the 2004 ACM symposium on Applied computing*, page 1366–1371, 2004. 2.4.2

[135] Rob Pike. Notes on programming in c. *URL http ://www. lysator. liu. se/c/pikestyle. html*, 1989. 2.3.4

[136] Mary Poppendieck and Thomas David Poppendieck. *Lean software development : an agile toolkit*. Addison-Wesley, Boston, Mass., 2003. (document), 0.3, 0.1, 0.3.1, 0.3.1, 5.1.2

[137] Eric Raymond. The cathedral and the bazaar (originally published in volume 3, number 3, march 1998). *First Monday*, 2005. 0.3.2, 2.5.4, 3.2

[138] Dzenan Ridjanovic. Modelibra software family. In *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, page 176–181. Springer, 2008. 1.1, 1.2.3, 1.3, 4, 4.2.2

[139] Dzenan Ridjanovic. Model driven prototyping with modelibra. In *Recent Trends in Wireless and Mobile Networks*, page 368–377. Springer, 2011. 1.1, 1.2.3, 1.3

[140] Winston W. Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26, 1970. (document), 0.2, 0.3.1, 2.5.1, 2.7

[141] James Rumbaugh. Object-oriented analysis and design (OOAD). 2003. 2.3.2, 2.3.4, 13

[142] Nayan B. Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3) :8, May 2010. 2.5.1, 2.5.1

[143] Douglas C Schmidt. Model-driven engineering MDE. *Computer*, 39(2) :0025–31, 2006. 0.3, 0.3.2, 2.3.6

[144] Michael Schrage. Never go to a client meeting without a prototype [software prototyping]. *Software, IEEE*, 21(2) :42–45, 2004. 0.3.1

[145] Sencha. Ext JS 4 Cookbook_Exploring further, 2012. 5.2.2

[146] Leon Shklar and Rich Rosen. *Web application architecture : principles, protocols, and practices*. John Wiley, Chichester, England ; Hoboken, NJ, 2003. 1.2.3

[147] Shriver. Measurable value with agile, 2008. 2.5.3

[148] Darius Silingas. Why MDA fails : Analysis of unsuccessful cases, 2013. 0.3.2, 3.2.1

[149] Herbert A Simon. *The sciences of the artificial*. MIT Press, Cambridge, Mass., 1996. 2.3.5, 2.3.6, 2.4.1

[150] PETR C. SMOLÍK. *Mambo Metamodeling Environment*. PhD thesis, Citeseer, 2006. 1.2.2, 3.1, 3.3, 3.3.1, 3.4, 7

[151] D. Spinellis. On the declarative specification of models. *IEEE Software*, 20(2) :96–95, March 2003. 0.3.2

[152] Peter Swithinbank. *Patterns : Model-Driven Development Using IBM Rational Software Architect*. IBM International Technical Support Organization, 2005. 0.3, 0.3.2, 3.2

[153] Leila Trabelsi and Inès Hammami Abid. Urbanization of information systems as a trigger for enhancing agility : A state in the tunisian firms. *European Journal of Business and Management*, 5(5) :63–77, 2013. 0.1

[154] Paul Valéry. *EUPALINOS ou l'Architecte*, volume 55. Gallimard, 1921. 5

[155] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 2000. 5

[156] Hans Van Vliet. *Software engineering : principles and practice*. Wiley, 3 edition, 2008. 00698. 1.2.1

[157] Michalis Vazirgiannis. Data Modeling—Object-Oriented data model. 2002. 2.3.4

[158] Mark Weiner, Micah Sherr, and Abigail Cohen. Metadata tables to enable dynamic data modeling and web interface design : the SEER example. *International journal of medical informatics*, 65(1) :51–58, 2002. 00010. 0.3.1, 1.1, 1.2.3, 1.3

[159] Sam Williams. *Free as in Freedom [Paperback] : Richard Stallman's Crusade for Free Software*. O'Reilly Media, Incorporated, 2012. 2.6

[160] David P Wood and Kyo C Kang. A classification and bibliography of software prototyping. 1992. 0.1

[161] Edward Yourdon. Just enough structured analysis. *Published at :< http ://www. yourdon. com*, 2006. 2.3.3

[162] JA Zachman. The challenge is change : A management paper. *Zachman International*, 1997. 2.4.2

[163] John A. Zachman. A framework for information systems architecture. *IBM systems journal*, 26(3) :276–292, 1987. 2.4.2, 2.5

[164] John A Zachman. Enterprise architecture : The issue of the century. *Database Programming and Design*, 10(3) :44–53, 1997. 2.4.2