CST –Analyse multi-facettes et opérationnelle pour la transformation des systèmes d'information

HOUEKPETODJI Mahugnon Honoré

1 Description du sujet de la thèse

CIM est une SAS au capital social de 200 000 euros détenu à 100% par DL Software. CIM est éditeur, intégrateur, hébergeur et infogéreur de solutions pour l'assurance de personnes en santé, prévoyance. Elle offre une expertise Santé et Prévoyance acquise après plus de 30 ans auprès de ses clients. CIM est hébergeur de ses solutions pour 90% de ses clients et plus de 1000 utilisateurs. Toutes les thématiques d'infrastructure et de surveillance des flux sont intégrées à cette offre. CIM est propriétaire de ses infrastructures serveurs, tous les éléments actifs des systèmes et tous les éléments de stockage sont achetés par CIM, gérés et supervisés par les équipes de CIM. Aucun sous-traitant n'intervient dans les opérations quotidiennes d'hébergement, d'exploitation des solutions et des données hébergées.

CIM est certifiée *Microsoft GOLD Partner*. Elle est l'éditeur des progiciels de la gamme Izy Links et assure l'intégration de l'ensemble des briques de cette gamme ainsi que des briques partenaires nécessaires à la bonne réussite du projet. Cette solution est développée en PowerBuilder sur une base de données DB2. L'équipe de développement vient d'avoir PowerBuilder version 2017 (été 2018) et est en cours de passage sur DB2 v11.

Le système de gestion est centré sur Izy Protect, autour duquel gravite l'ensemble des briques complémentaires répondant à l'assemble des besoins, et pouvant être activées ou non. La société CIM a effectué une analyse de risque pour son évolution et croissance en 2017. D'où il ressort que Izy Protect souffre des problèmes (1) vieux langage, (2) logiciel vieillissant, (3) perte de savoir, (4) changements à haut risque. Ces problèmes sont récurrents chez les organismes gérant des systèmes d'information [14].

Face à cette situation CIM souhaite entreprendre des actions en particulier migrer la partie métier d'Izy Protect vers *C#*. Ce travail de doctorat consiste à proposer des modèles et des mécanismes d'aide pour une ré-ingénierie des systèmes d'information. Par exemple, des mécanismes et des outils permettant d'aider les développeurs de la CIM à migrer la partie métier d'Izy Protect vers *C#*. Les expériences et validation des prototypes se feront dans le contexte de l'application du système d'information écrit en PowerBuilder de la société CIM.

2 Etat de l'art

Dans cette section, je presenterai Izy Protect et les travaux qui se penchent sur la question de la ré-ingénierie des systèmes d'information.

2.1 Présentation d'Izy Protect

Izy Protect est un système de plus de 3 millions de lignes de code écrites en PowerBuilder et maintenu depuis plus de 20 ans par les développeurs de la CIM. Aujourd'hui Izy Protect est maintenu par une équipe de 18 développeurs. Le code source est organisé par bibliothèques PowerBuilder. Izy Protect compte 117 bibliothèques. Les bibliothèques d'Izy Protect peuvent être regroupées en des modules métier. Par contre cette information ne peut pas être directement déduite du code. La plus large a une taille d'environ 300 000 lignes de code. Durant toutes ces années, il y a eu beaucoup de changements dans l'équipe de développeurs. Vu la complexité actuelle du système, les développeurs ont de plus en plus du mal à le maintenir. Les anciennes versions du système sont stockées sur un disque dur. Pour des raisons internes à la CIM, les versions d'Izy Protect ont été perdues jusqu'en 2010. De plus, les développeurs risquent à tout moment de faire de régression (casser une fonctionnalité existante). Pour cause, le code source de Izy Protect n'est pas couvert avec des tests unitaires automatisés. Ceci augmente la craint des développeurs pour de petites modifications. Les caractéristiques d'Izy Protect montrent qu'il est un système qui a une grande valeur pour l'entreprise et il est développé avec un vieux langage de

programmation : PowerBuilder. Par conséquent, il est **patrimonial** selon Demeyer et al. [14]. Quand un système a plusieurs décennies de vie, la rétro-ingénierie est une activité centrale pour le maintenir [14].

Dans l'entreprise, les travaux de maintenances ou de développements sur Izy Protect sont identifiés par des tickets. Les tickets sont stockés dans la base de données des tickets ou fiches navettes depuis 1998. La base de données des tickets pilote l'ensemble du processus d'évolution du logiciel : attribution du travail aux développeurs, gestion du flux de travail pour répondre à une demande du client, informations de facturation sur chaque tâche. Il existe des tickets pour la correction de défauts, la rédaction de documentation, l'ajout de nouvelles fonctionnalités, etc.

Un ticket comporte entre autres les caractéristiques suivantes :

— la date de création
— la date de clôture
— l'estimation du temps nécessaire au développeur pour travailler sur le ticket
— temps passé par un développeur
  — le temps d'analyser
  — le temps de mettre en œuvre une solution
  — le temps de test
— le(s) bibliothèque(s) impactée(es)

Izy Protect est planifié pour les tests de régressions lorsque les modifications concernent certaines parties du système présélectionnés par l'entreprise. Il faut noté que ces parties sont statiques. Les tests de régressions sont subdivisés en deux parties : les tests de traitements de données(tests fonctionels) et les tests IHMs. Les tests de traitements de données et les test IHMs sont automatisés. Les tests de données sont des tests *black box* : on passe des données en entrée par *Bash*, on suit l'exécution et on compare la sortie. Tandis que les tests IHMS sont fait avec l'outil Tosca [4]. Par contre, au cas où les modifications concernent d'autres parties du système, il n'y a aucune assurance qu'il n'a pas de régressions ce qui peut parfois se faire sentir chez le client.

2.2 Les tests automatisés et le DevOps

Dudekula Mohammad Rafi et al. [18] ont étudié les avantages et les limitations de l'automatisation des tests. Ils ressortent que les tests automatisés sont répétables et réutilisables. Les tests automatisés assurent aussi une couverture du code et un gain de temps dans l'exécution des tests. Mais en contrepartie, les tests automatisés coûtent pour la mise en place(sélection d'outil, achat de l'outil de tests, conception des cas de tests), formations du personnel.

Wang et al. [37] ont présenté une étude qui examine le degré de maturité du processus d'automatisation des tests par rapport aux pratiques d'automatisation des tests adopter dans ces entreprises de logiciels. Ils remarquent que la culture du DevOps favorise un haut degré de maturité du processus d'automatisation des tests.

Diaz et al. [15] ont enquêté sur pourquoi les entreprises installent la culture du DevOps dans leurs organisations. Les résultats montrent que les entreprises avaient besoin de : être plus agile et rapide, répondre aux tendances du marché, livrer les produitsservices de meilleure qualité, réduire le temps de configuration des environnements, améliorer la communication entre les équipes, introduire le processus d'automatisation et amener les commerciaux, les développeurs, et les opérateurs à avoir un objectif fonctionel commun. L'installation du DevOps à apporter : une rapidité de livraison, une meilleure qualité des produits et services, automatisation des processus, améliorer la collaboration et la communication entre les équipes et une meilleure concentration sur les besoins du client.

2.3 Analyse de l'évolution de l'état d'un système logiciel patrimonial

Les systèmes patrimoniaux sont des systèmes en constant changement : production de nouvelles fonctionnalités. La deuxième loi de Lehman [24] stipule qu'à mesure que les logiciels évoluent, la complexité croissante et l'augmentation des défauts entraîneront une baisse de la satisfaction des parties prenantes, à moins que les équipes de projet n'entreprennent le travail nécessaire pour maintenir la qualité. Selon Demeyer et al. [14], collecter et interpréter les données et les résumer dans une vue cohérente est une étape importante pour la compréhension initiale d'un système logiciel existant. Dans ce sens, de nombreux travaux de la littérature proposent des techniques, pour suivre l'évolution de l'état de ces systèmes.

Zhang and Kim [38] utilise les données des occurrences bugs et le temps pour modéliser l'évolution d'un système logiciel avec ccharts.

Lenarduzzi et al. [26] propose un système de recommandation d'action au développeur pour un nouveau bug. Le système se base sur l'historique des bugs, le code source ainsi que qu'un algorithme de prédiction. Pour que ça marche, le code source doit être géré dans un système de contrôle de versions. Ce qui n'est pas le cas avec Izy Protect.

Port and Taber [31] utilise les modèles et l'analyse de l'historique des défauts pour évaluer la qualité d'un système et prédire l'effort nécessaire pour améliorer le système. Les métriques mesurées sont : le taux de bugs sur une période de

temps, le ratio entre le taux d'augmentation de la taille du système et le taux de bugs, le temps moyen entre la découverte des bugs, l'effort pour résoudre les bugs, estimation des risques de futurs bugs... Certain de ces métriques comme : le taux de bugs par période de temps, l'effort pour résoudre les bugs sont intéressants dans le cadre d'Izy Protect. Les autres ne se conforment pas au contexte, car les tickets d'Izy protect ne sont pas directement liés au code de façon standard. Chaque développeur note le numéro de tickets à sa façon dans le code.

Bibi et al. [9], Kim et al. [23] proposent des modèles de prédiction des défauts avec des algorithmes d'apprentissage en utilisant l'historique des bugs de système. Les algorithmes de prédiction de bugs ne sont pas toujours consistants [6]. De plus, ils ne tiennent pas compte des changements qui peuvent être imprévisible dans le code. En plus, Izy Protect est logiciel commercial, multi-utilisateur, et donc chaque utilisateur a des fonctionnalités ou des changements qui lui sont spécifiques.

Nagappan and Ball [27] utilise l'historique le nombre de lignes de code changées pour une correction de bug ou une nouvelle fonctionnalité pour prédire la densité de bug dans le code. Pour que ceci soit réalisable, il faut que le code soit préalablement versionné dans un système de contrôle de versions.

Raja et al. [32] a étudié différents algorithmes de modélisation des défauts d'un système à partir des données des défauts relevés sur huit projets dont les codes sources sont ouverts. Il en ressort que la moyenne glissée modélise mieux les défauts des systèmes.

2.4 Rétro-ingénierie

Dans cette section, je présenterai les travaux reliés à l'outillage pour la rétro-ingénierie que j'ai étudiée. Chikofsky and Cross II [11] définissent la rétro-ingénierie comme *un processus d'analyse d'un système donné pour identifier les composants du système et leurs relations, afin de créer des représentations du système sous une autre forme ou à un niveau d'abstraction plus élevé."*.

Bruneliere et al. [10] affirment que la rétro-ingénierie n'est pas limitée à certains langages courants, mais est universelle. Par exemple, elle peut concerner la base de données [13] ou l'interface graphique [35]. Il est donc nécessaire de disposer d'une suite d'outils polyvalents et extensibles, indépendants du langage : cette extension peut se faire à plusieurs niveaux - métamodèle, mais aussi au niveau des outils eux-mêmes (par exemple en agissant sur le modèle).

C'est dans cette optique que Kienle and Müller [22] énoncent la retro-ingénierie nécessite des outils qui fournissent des fonctionnalités permettant d'extraire des informations de bas niveau des systèmes, d'analyser et de générer des connaissances sur les systèmes, et de visualiser ces connaissances afin que les ingénieurs puissent comprendre efficacement les aspects du système qui les intéressent. De ce point de vue ils identifient que ces outils doivent être :

(1) scalable : capacité de ces outils à s'adapter au changement de taille du logiciel ; (2) interoperable : ces outils doivent être capable de communiquer avec des outils externes ; (3) personnalisable : les activités de rétro-ingénierie étant variables, les utilisateurs d'outils de rétro-ingénierie doivent pouvoir continuellement les adapter pour répondre aux besoins changeants ; (4) utilisable : facilité d'utilisation ; (5) adoptable : facilité d'apprentissage.

Bellay and Gall [8] ont proposé un ensemble de critères pour comparer les outils de rétro-ingénierie. Par contre, ils ne couvrent pas le métamodèle, l'extension du métamodèle et des outils de manière exhaustive.

Govin [19] ont identifié des critères pour les outils de réingénierie. Parmi ces critères, j'ai retenu ceux qui se rapportent à la rétro-ingénierie. Ce sont les critères (1) de sélection (2) d'abstraction. Le critère de sélection exige que l'outil de rétro-ingénierie permette à l'utilisateur de sélectionner des éléments du code source d'un système qui répondent à une requête donnée. L'outil doit aussi permettre à l'utilisateur d'abstraire les caractéristiques des entités du code source à un haut niveau d'abstraction. Par exemple : déduire de la complexité cyclomatique des classes d'un package, la complexité cyclomatique du package.

Bruneliere et al. [10] ont proposé et mis en œuvre le cadre MoDisco. MoDisco est divisé en 4 couches, un projet de modélisation de la plateforme Eclipse, l'infrastructure, la technologie et les cas d'utilisation. La couche infrastructure contient des outils génériques permettant de naviguer et d'interroger les modèles. La couche technologie contient le métamodèle spécifique (Java, XML, JSP, ...). Et la couche des cas d'utilisation contient l'action spécifique pour un modèle, par exemple le remaniement de code Java. MoDisco inclut : un navigateur de modèle, un éditeur pour voir le code des entités d'un modèle, un support graphique pour faire des requêtes sur les éléments du modèle. Par contre Bruneliere et al. [10], ne propose que l'UML pour visualiser les éléments d'un modèle. Alors que pour un logiciel patrimonial, Izy Protect par exemple, l'UML devient vite une toile d'araignée et freine une compréhension rapide du système.

3 Avancées actuelles

3.1 Route vers le DevOps

Izy Protect présente les problèmes cités dans la Section 2.1 en particulier, code non versionné, ce qui engendre les pertes et écrasement de code, puis l'absence totale de tests uintaires automatisés qui couvrent les fonctionnalités au niveaux du code source, etc. Mon objectif est de construire des outils visant à aider à la réingénierie de Izy Protect.

Par ailleur, Demeyer et al. [14] affiment que les tests sont comme une assurance de vie dans le cadre de la réingénierie logicielle. Ils vont plus loin pour dire que les tests permettent de garantir les activités de réingénieries ne vont pas modifier la logique metier. Sachant que la réingénierie de Izy Protect se fera de façon incrémentale, en cas d'erreur, il est important de savoir quel changement a cassé le code. Un système de contrôle de versions et les tests automatisés seront d'une grande utilité pour les activités de réingénieries. De plus, le fait de versionner le code non seulement permet d'éviter les problèmes de pertes de code mais aussi à long terme me permettra aussi d'avoir une base d'analyse de l'évolution du code d'Izy Protect suplémentaire et de fournir un outil de vérification des normes de développements sur les changements effectués par les développeurs.

J'ai choisi Subversion (SVN) [3] par le fait de sa simplicité pour la compréhension. Il faut aussi noté que le module de contrôle de versions qu'offre PowerBuilder n'est pas trop stable. Par exemple, parfois, le module considère un code déjà versionné comme un code non versionné. De plus, le module de contrôle de versions qu'offre PowerBuilder ne supporte pas bien les opérations avancées comme la comparaison de deux versions du code ou bien la résolution de conflit. Tout cela fait de SVN un choix simple pour commencer. En complément de PowerBuilder, j'ai amené les développeurs de la CIM à utiliser TortoiseSVN [1]. Il faut signaler que pour le succès de la mise en place, il a fallu une documentation et une assistance continue.

Dans le but d'avoir une base d'analyse de l'évolution du code d'Izy Protect, je suis en train de reconstruire l'histoire du code source d'Izy Protect depuis 2012 afin de procéder à des analyses de l'évolution du système à partir des changements entre les versions. Les versions d'Izy Protect sont nombreuses. J'en suis actuellement sur les versions produites en 2015.

En ce qui concerne les tests unitaires automatisés,PBUnit[2] paraît être la librairie qui permet de tester les fonctionnalités des applications PowerBuilder au niveau du code source.

3.2 Analyse des fiches navettes

Afin d'évaluer l'état d'Izy Protect, et contrôler l'effet de la ré-ingénierie sur Izy Protect, j'ai utilisé la base de donnée des fiches navettes. A travers l'analyse des tickets, j'espère caractériser l'état d'Izy Protect avec des données factuelles, et suivre les changements du système dans le temps. Ceci permettra d'evaluer l'impact de l'utilisation des outils d'aide à la rétro-ingénierie que je proposerai durant cette thèse. Pour cela, j'ai utilisé la base de donnée des fiches navettes. Après nettoyage, seuls les tickets à partir de 2004 sont utilisables. La Table 1 montre un recapitulatif des tickets analysés. Il

TABLE 1 – Tickets

	correction	évolution	Total
Tickets	15407	11973	27380
% temps	28%	72%	100%

indique que 28% du temps passé est pour des corrections tandis que 72% est pour des évolutions. Selon Pigoski [30] la proportion de tickets de corrections doit être entre 20% et 25% . Par conséquent, la proportion de tickets de corrections sur Izy Protect paraît élevée.

En me basant sur les résultats de [32], j'ai utilisé la moyenne glissante avec une porté de cinq mois pour modéliser l'état du système. La Figure 1 présente un tableau de bord qui résume les resultats.

En effet, sur la Figure 1 le premier graphe à gauche montre le temps nécessaire aux développeurs pour traiter les tickets. On voit que ce temps augmente.

Ceci pourrait être une conséquence de la complexité du code d'Izy Protect, ou bien, une mauvaise compréhension des besoins du client qui fait que le développeur prend du temps a comprendre le travail à faire. Le premier graphe à droite sur la Figure 1 montre le temps des tests manuels du développeur. Contrairement au temps pour traiter les tickets, les développeurs passent de moins en moins de temps pour tester leur code. Le deuxième graphe à gauche sur la Figure 1 montre le temps de test par l'équipe de ressource et qualité (RC). Ce temps augmente. Ceci pourrait être dû à la quantité de données traitées par les tests ou bien, la complexité d'Izy Protect qui peut être responsable de la complexité des cas de tests. Le fait que les développpeurs testent de moins en moins Izy Protect pourrait engendrer beaucoup de va-et-vient d'Izy Protect entre l'équipe de RC et des développeurs (le cercle -développement, RC : bugs, correction, RC)

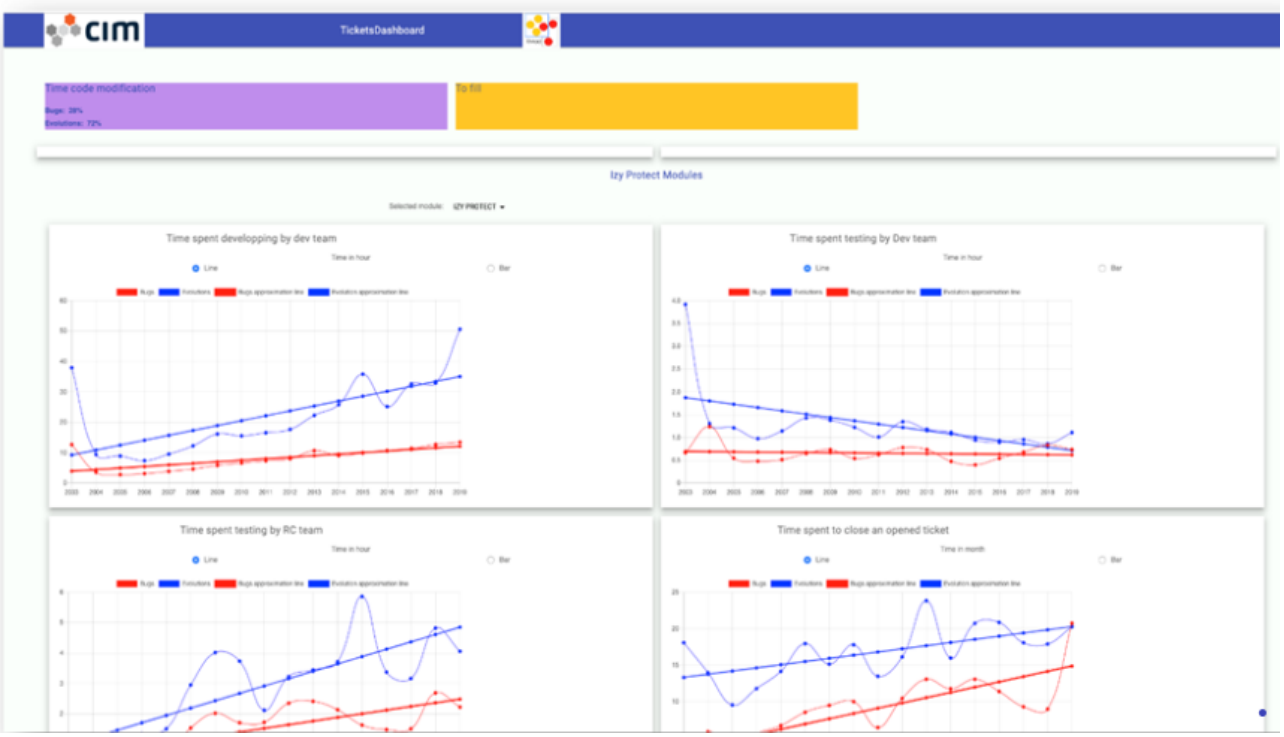

FIGURE 1 – Tableau de bord d'analyse de tickets

Le deuxième graphe à droite montre le temps de fermeture de tickets. Ce temps est aussi en augmentation.

Le tableau de bord se met à jour une fois par mois. Donc je peux continuellement mesurer l'impact de l'utilisation des outils d'aide à la rétro-ingénierie que je proposerai.

3.3 Outil d'aide à la rétro-ingénierie logicielle

Pour répondre aux exigences détaillées dans la Section 2.4, j'ai développé une suite d'outils d'aide à la rétro-ingénierie. Ces outils sont développés au-dessus de la plateforme Moose [28]. En effet, la plate-forme offre un métamodèle générique et quatre outils principaux pour l'analyse des systèmes logiciels. Il s'agit de (1) Famix : un meta-modèle qui permet aux développeurs de représenter un programme, (2) Moose Query : un API pour naviguer dans un modèle Famix, (3) Les tags : utilisés pour enrichir le modèle de l'application avec des informations qui ne peuvent pas être directement déduites du code source et (4) Roassal : un framework de visualisation intégré dans Moose.

Dans la suite, je présenterai d'abord l'architecture mise en place pour les outils puis chaque outil.

3.3.1 Architecture des outils

La Figure 2 montre l'architecture globale de la suite d'outils. L'architecture globale de la suite d'outils est principalement composée d'un *BrowserMaster* et des navigateurs. Le tout s'exécute sur une instance d'un modèle Famix. Il a été pensé dans le but de faciliter l'expérience du développeur dans les différentes activités de la rétro-ingénierie logiciel. Le *BrowserMaster* est responsable de l'échange d'information entre les navigateurs. Il est au courant de tous les navigateurs ouverts. Il se charge de notifier tous les navigateurs ouverts en cas d'événement.

Tous les navigateurs partagent le même contexte. Quand une entité est selectionnée, chaque navigateur la montre de son propre point de vue.

3.3.2 Navigateur de graphe d'appel

La Figure 3 donne un aperçu du graphe d'appel avec les options. La fenêtre (1) de cet outil permet principalement de visualiser sous forme d'un graphe les entités qui utilisent l'entité courante du navigateur. Les nœuds représentent les entités. Les flèches représentent l'ensemble des dépendances entre deux entités. Le sens de la flèche indique le sens des dépendances. En effet pour un système logiciel comme Izy Protect par exemple, ce graphe peut rapidement devenir illisible. Pour palier à ce problème, l'outil intègre un panel d'option (la fenêtre (2) de la Figure 3) qui permet de filtrer le graphe

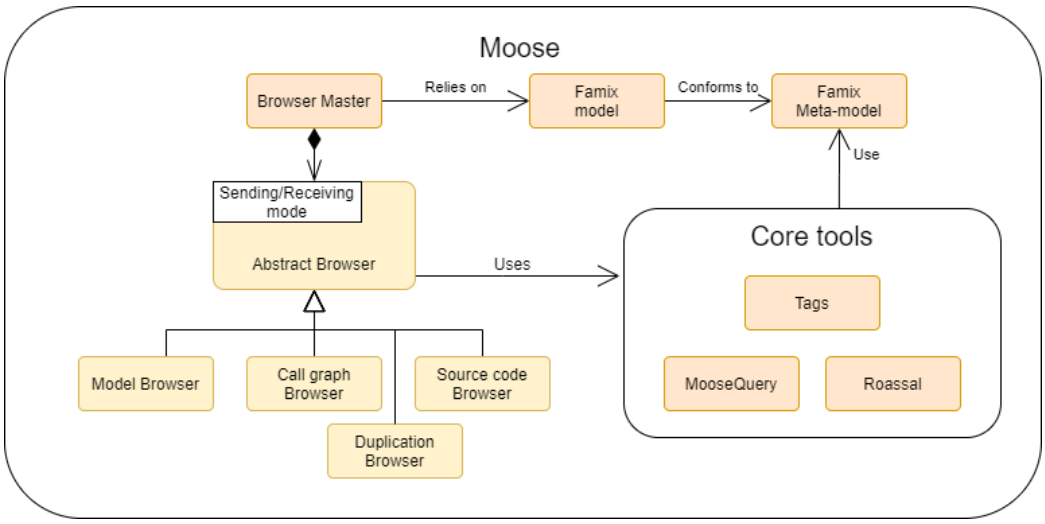

FIGURE 2 – Architecture de la suite d'outils

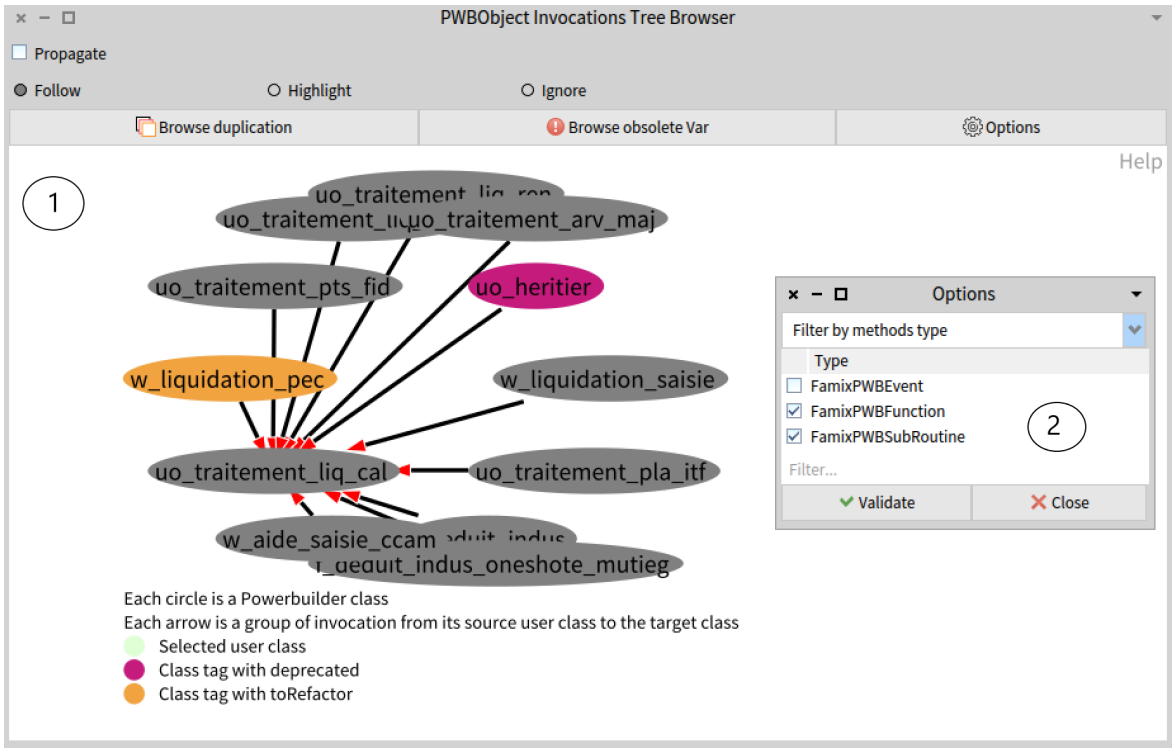

FIGURE 3 – Graphe d'appel avec l'option de reduction de complexité du graphe

par type d'entité à l'origine des appels. Afin de donner plus de contexte aux développeurs, quand il glisse la souris sur une flèche, un pop-up lui montre le code source de toutes les dépendances que la flèche regroupe.

Le navigateur de graphe d'appel permet aussi aux développeurs de marquer les entités afin de leurs ajouter une information qu'on ne peut pas extraire directement du code source. En d'autres termes, une connaissance qui ressort de l'expérience du développeur sur le système.

3.3.3 Navigateur de Code mort

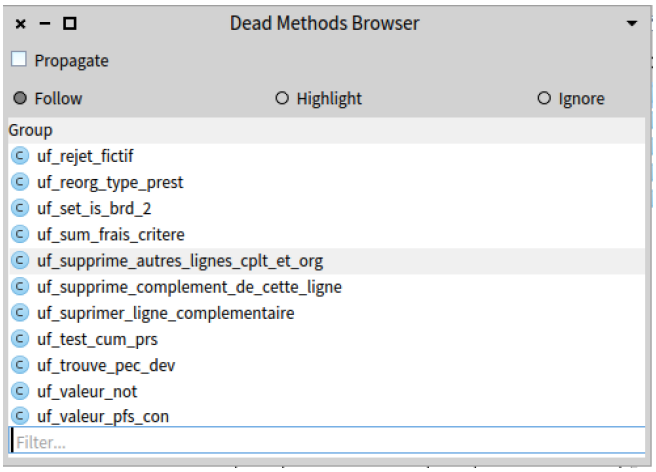

FIGURE 4 – Navigateur de code mort

Un code mort est une partie du code source du programme qui n'est jamais utilisée lors de l'exécution de ce dernier. Il peut être un obstacle à la compréhension d'un programme dans les activités de réto-ingénieries [36]. Malheureusement le code mort est identifié pour être généralement présent dans les logiciels [33]. Par conséquent, Izy Protect contient probablement de code mort. Pour faciliter la compréhension de Izy Protect, il faut donc lister et enlever du code mort . La Figure 4 montre le navigateur de code mort. C'est l'outil que je propose pour aider les développeurs à enlever le code mort. Ce navigateur présent pour le moment les méthodes de l'entité courante du navigateur qui ne sont jamais appelées dans le système ainsi que les méthodes qu'elles appellent.

3.3.4 Navigateur de code dupliqué

D'après la littérature, (5-10%) du code source des programmes de grande taille sont des clones [7]. Détecter et enlever ces clones réduit le coût de maintenance de ces systèmes [7]. Dans le contexte d'Izy Protect, pour la migration de sa partie métier, le fait de connaître les clones facilitera la migration dans le sens où les développeurs pourront migrer seulement une fois les clones ou bien le remettre facilement. La Figure 5 présente le navigateur de code dupliqué. Les carrés externes représentent les entités qui contiennent des clones. Dans le cas de la Figure 5, ces entités sont des méthodes. Les carrés internes représentent les clones que présente une entité. Les clones sont représentés de telle façon que l'utilisateur puisse facilement les inspecter. Le navigateur de code dupliqué utilise actuellement un algorithme de détection basé sur l'égalité stricte des chaînes de caractères [16]. Cet algorithme peut être remplacé par un algorithme plus sophistiqué pour détecter les doublons [34].

Sur la Figure 5, la troisième entité sur la première ligne ($uf\_calcul\_sig\_cas\_exp\_pos$) et la deuxème entité sur la deuxième ligne ($uf\_calcul\_sig\_cas$) ont du code dupliqué entre eux. Quand l'utilisateur clique sur $uf\_calcul\_sig\_cas\_exp\_pos$, ces clones (carrées internes) prennent des couleurs différentes. Les clones que $uf\_calcul\_sig\_cas\_exp\_pos$ a en commun avec $uf\_calcul\_sig\_cas$, dans $uf\_calcul\_sig\_cas$ prennent les mêmes couleurs que leurs semblables dans $uf\_calcul\_sig\_cas\_exp\_pos$. Par exemple le carré bleu dans $uf\_calcul\_sig\_cas$ a pris la couleur du carré bleu dans $uf\_calcul\_sig\_cas\_exp\_pos$ parceque les deux carrées représentent le même fragment de code. Cela permet de voir plus facilement quelles entités ont des clones et de comparer les codes sources dans le navigateur de code que je présenterai dans la suite.

3.3.5 Navigateur de code source

La Figure 6 présente le navigateur de code source. Ce navigateur est une fenêtre qui affiche le code source de l'entité courante. En particulier dans le cadre d'un fragment dupliqué, le code source de l'entité qui contient ce fragment est affiché

FIGURE 5 – Navigateur de code dupliqué

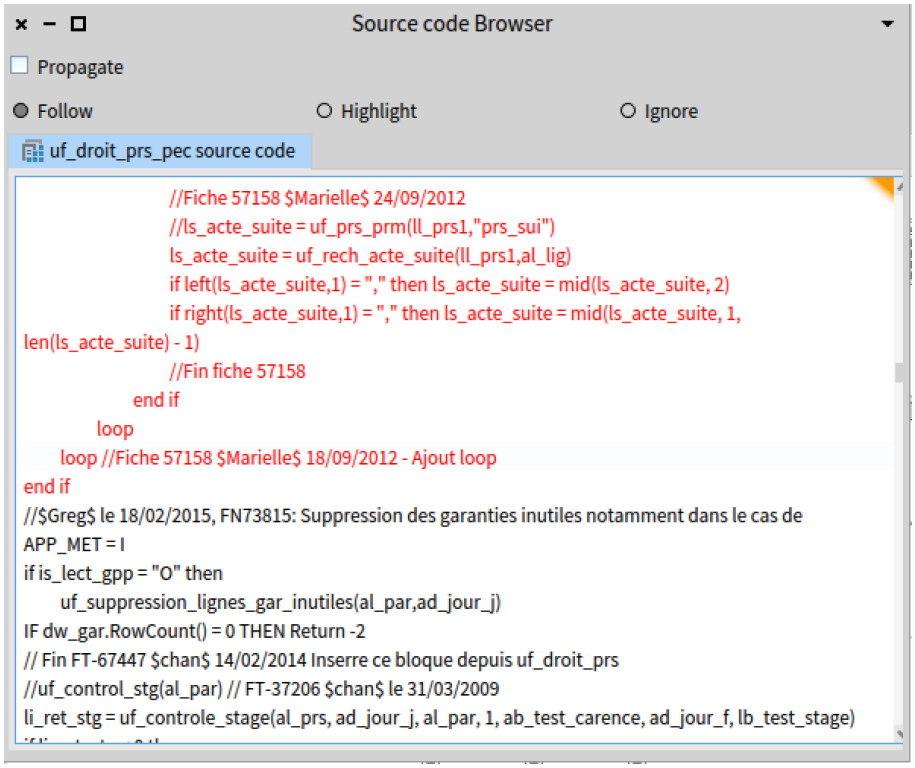

FIGURE 6 – Navigateur de code dupliqué

avec la partie du code qui représente le fragment dupliqué en rouge comme sur la Figure 6.

3.3.6 Conformité des navigateurs par rapport aux critères des outils de rétro-ingénieurie

J'ai construit les navigateurs de façon a ce qu'ils respectent les critères précités dans la Section 2.4. En effet, l'environnement Moose offre les propriétés de **scalablilité**, **d'interopérabilité** [17] dont héritent les outils. À travers les différentes réunions de présentations des outils aux développeurs, j'ai pris en compte leurs critiques afin que les outils non seulement, répondent à leurs besoins, mais aussi, soient facile d'utilisation et d'apprentissage. Le *Navigateur de graphe d'appel* Section 3.3.2 répond aux critères **d'abstraction** car il permet par exemple d'abstraire l'ensemble des invocations des méthodes d'une classe au niveau de la classe.

4 Travaux futurs

Tous les outils présents ci-dessus sont conçus dans le but de nettoyer le code et de visualiser les interactions entre les différentes classes d'Izy Protect. Pour vérifier que les modifications n'introduisent pas d'erreurs, j'envisage de mettre en place les tests automatisés avec PBUnit. Ceci m'amènera à étudier les différentes approches de tests abordés dans la littérature afin d'identifier une approche qui s'adapte au contexte d'Izy Protect. Toutefois je pense que transformer les tests de données énoncés dans la Section 2.1 en tests automatisés sera un point de départ.

L'entreprise prévoit de migrer une partie du système dans une architecture orienté service. Les développeurs ont donc exprimé le besoin d'extraire des logiques métiers du code. Plusieurs travaux sont présentés dans la littérature dans ce sens notamment [12, 25, 29] qui se basent sur l'analyse statique des interactions des fonctions avec les variables pour extraire les logiques metiers d'un système patrimonial. Je souhaite combiner cette méthode avec la méthode proposée [5] pour proposer un outil d'extraction des logiques métiers dans les systèmes patrimoniaux. Pour mener à bien la migration, je proposerai un outil de génération de tests unitaires automatisés pour les parties de Izy Protect concernées.

5 Publications

Dans le cadre de cette thèse,
— l'article : *Improving practices in a medium French company : First step* (Honore et al. [21]) a été soumis
— l'article : *Towards a Versatile Reverse Engineering Tool Suite* (Honore et al. [20]) a été accepté en version court.

6 Formations

Pour les formations doctorales, je cumule un total de 41 crédits sur les 40 cédits minimums demandés pour une thèse CIFRE. Voici la liste des formations auquelles j'ai assistées avec leurs crédits.
— Les fondamentaux du management d'équipe Session 1 (Ecole doctorale : 7 crédits)
— Gestion de conflit (CIM)
— Formation Propriété intellectuelle au service des doctorants tronc commun (Ecole doctorale : 10 crédits)
— Intelligence économique et dynamique de l'innovation (Ecole doctorale : 10 crédits)
— Communiquer en Anglais - Niveau confirmé - Stage intensif (Ecole doctorale : 14 crédits)

Bien que j'ai cumulé le nombre de crédits minimums demandés, j'assisterai à d'autres formations ayant trait avec le développement personnel. Car c'est un sujet qui m'intérresse.

7 Projet professionnel

En ce qui concerne mon projet professionnel, je souhaite continuer dans l'enseignement supérieur : donner des cours et continuer dans la recherche. Je pense que la réingénierie des systèmes logiciels est un axe de recherche où beaucoup de travaux intéressant sont menés. Néanmoins il reste à faire et je souhaite contribuer à cela. Toutefois, je ne me refuse pas l'idée de démarrer une start-up à l'issue de ma thèse ou travailler dans une entreprise.

Références

[1] Tortoisesvn. https://tortoisesvn.net/.

[2] Pbunit. https://sourceforge.net/p/pbunit/wiki/Home/.

[3] Subversion. https://subversion.apache.org/.

[4] Tosca testing. https://www.tricentis.com/products.

[5] Nicolas Anquetil, Anne Etien, Gaelle Andreo, and Stéphane Ducasse. Decomposing God Classes at Siemens. In *International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, United States, October 2019. URL https://hal.inria.fr/hal-02395836.

[6] Abdul Ali Bangash, Hareem Sahar, Abram Hindle, and Karim Ali. On the time-based conclusion stability of software defect prediction models. *arXiv preprint arXiv :1911.06348*, 2019.

[7] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 368–377. IEEE Computer Society, Washington, DC, USA, 1998. doi: 10.1109/ICSM.1998.738528.

[8] Berndt Bellay and Harald Gall. An evaluation of reverse engineering tools. *Journal of Software Maintenance : Research and Practice*, 1998.

[9] S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahvas. Software defect prediction using regression via classification. In *IEEE International Conference on Computer Systems and Applications, 2006.*, pages 330–336, 2006.

[10] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco : A model driven reverse engineering framework. *Information and Software Technology*, 56(8) :1012–1032, 2014.

[11] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery : A taxonomy. *IEEE Software*, 7(1) :13–17, January 1990. doi: 10.1109/52.43044. URL http://dx.doi.org/10.1109/52.43044.

[12] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. A model driven reverse engineering framework for extracting business rules out of a java application. In Antonis Bikakis and Adrian Giurca, editors, *Rules on the Web : Research and Applications*, pages 17–31, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32689-9.

[13] Julien Delplanque, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Recommendations for evolving relational databases. In *32nd International Conference on Advanced Information Systems Engineering*, 2020.

[14] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. ISBN 1-55860-639-4. URL http://rmod.inria.fr/archives/books/OORP.pdf.

[15] Jessica Diaz, Daniel López-Fernández, Jorge Perez, and Ángel González-Prieto. Why are many business instilling a devops culture into their organization ?, 2020.

[16] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society, September 1999. doi: 10.1109/ICSM.1999.792593. URL http://rmod.inria.fr/archives/papers/Duca99bCodeDuplication.pdf.

[17] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose : a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005. ISBN 88-464-6396-X. URL http://rmod.inria.fr/archives/papers/Duca05aMooseBookChapter.pdf.

[18] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing : Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012.

[19] Brice Govin. *Support à la rénovation d'une architecture logicielle patrimoniale : Un cas réel chez Thales Air Systems*. PhD thesis, Université de Lille, June 2018. URL http://rmod.inria.fr/archives/phd/PhD-20180-GovinBrice.pdf.

[20] HOUEKPETODJI Honore, Verhaeghe Benoît, Clotilde Toullec, Derras Mustapha, Fatiha Djareddir, Jerôme Sudich, Ducasse Stéphane, and Etien Anne. Towards a versatile reverse engineering tool suite. In *QUATIC 2020 : 13th International Conference on the Quality of Information and Communications Technology*, Faro, Portugal, 2020.

[21] HOUEKPETODJI Honore, Fatiha Djareddir, Jerôme Sudich, and Nicolas Anquetil. Improving practices in a medium french company : First step. In *RIMEL : Journée de travail Rimel / Lignes de produit / Sécurité*, Paris, France, 2020.

[22] Holger M. Kienle and Hausi A. Müller. The tools perspective on software reverse engineering : Requirements, construction, and evaluation. In *Advanced in Computers*, volume 79, pages 189–290. Elsevier, 2010.

[23] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498. IEEE, 2007.

[24] Manny Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

[25] Lei Wu, H. Sahraoui, and P. Valtchev. Coping with legacy system migration complexity. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 600–609, 2005.

[26] Valentina Lenarduzzi, Alexandru Cristian Stan, Davide Taibi, Davide Tosi, and Gustavs Venters. A dynamical quality model to continuously monitor software maintenance. In *The European Conference on Information Systems Management*, pages 168–178. Academic Conferences International Limited, 2017.

[27] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.

[28] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose : an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press. ISBN 1-59593-014-0. doi: 10.1145/1095430.1081707. URL http://rmod.inria.fr/archives/papers/Nier05cStoryOfMoose.pdf. Invited paper.

[29] Kestutis Normantas and Olegas Vasilecas. Extracting business rules from existing enterprise software system. In *International Conference on Information and Software Technologies*, pages 482–496. Springer, 2012.

[30] Thomas M. Pigoski. *Practical Software Maintenance : Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996. ISBN 0471170011, 9780471170013.

[31] Dan Port and Bill Taber. Actionable analytics for strategic maintenance of critical software : an industry experience report. *IEEE Software*, 35(1) : 58–63, 2017.

[32] Uzma Raja, David P. Hale, and Joanne E. Hale. Modeling software evolution defects : a time series approach. *Journal of Software Maintenance and Evolution : Research and Practice*, 21(1) :49–71, 2009. doi: 10.1002/smr.398. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.398.

[33] S. Romano. Dead code. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 737–742, 2018.

[34] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115) :64–68, 2007.

[35] Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse, and Mustapha Derras. Gui migration using mde from gwt to angular 6 : An industrial case. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019. URL https://hal.inria.fr/hal-02019015.

[36] X. Wang, Y. Zhang, L. Zhao, and X. Chen. Dead code detection method based on program slicing. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 155–158, 2017.

[37] Yuqing Wang, Mika V. Mäntylä, Serge Demeyer, Kristian Wiklund, Sigrid Eldh, and Tatu Kairi. Software test automation maturity – a survey of the state of the practice, 2020.

[38] H. Zhang and S. Kim. Monitoring software quality evolution for defects. *IEEE Software*, 27(4) :58–64, July 2010. ISSN 1937-4194. doi: 10.1109/MS.2010.66.