



AVERTISSEMENT

CE DOCUMENT EST UN RAPPORT DE STAGE de juin 2016

IL A ETE EDITE A L'ATTENTION DES STAGIAIRES DU DUT INFORMATIQUE DE VANNES.

CERTAINS NOMS, CERTAINES ILLUSTRATIONS ET CERTAINES DONNEES ONT ETE MODIFIES. IL N'A DONC PAS VOCATION A SERVIR DE DOCUMENTATION TECHNIQUE.

LES ETUDIANTS PEUVENT S'EN INSPIRER POUR LA REDACTION DE LEURS FUTURS RAPPORTS. Ils seront attentifs à :

- l'ordre des différentes parties (remerciements, table des matières, corps du rapport, conclusion, glossaire, table des figures, résumé...)
- la pagination
- le titrage (pas de titres générique/passe-partout comme « solution technique trouvée »)

DES COURS DE COMMUNICATION, EN 2^{ème} ANNEE DE DUT, SONT CONSACRES AU RAPPORT DE STAGE. LES ETUDIANTS SONT INVITES A S'Y REPORTER.

Jean-François KAMP et Hélène TUFFIGO



LE POLLES--POTIN Léandre

Année universitaire 2015-2016

DEPARTEMENT INFORMATIQUE
Institut Universitaire de Technologie
8, rue Montaigne
BP 561
56 017 Vannes CEDEX

Rapport de STAGE de fin d'études

« Développement d'un outil de validation de
faisabilité de déploiement »

Stage effectué au sein de l'entreprise

Capgemini



Du 10 avril au 27 juin 2016

Maitre de stage :

M. Pierre-Antoine BONNIN, Architecte Solutions S
Entreprise Capgemini, 35510 Cesson-Sévigné

Enseignant tuteur, IUT de Vannes :

M. Jean-Luc EVENO

Remerciements

Je tiens tout d'abord et tout particulièrement à remercier Pierre-Antoine Bonnin, mon maître de stage au sein de l'entreprise. Grâce à lui j'ai pu effectuer un stage de qualité dans une des entreprises informatiques les plus renommées au monde et je lui en suis sincèrement reconnaissant.

Je tiens à le remercier chaleureusement pour le temps consacré à l'encadrement de mon stage. J'ai pu constater la charge de travail à laquelle il était soumis, mais cela ne l'a jamais empêché de répondre à mes questions. Au-delà de ça, il a également fait en sorte de m'enseigner beaucoup sur la conception et la programmation informatique. Ce stage n'a donc pas été uniquement une manière de confronter mes connaissances à une situation concrète, mais il a également été un terrain d'apprentissage.

Je remercie l'équipe pédagogique de l'IUT de Vannes pour l'aide à la préparation de ce stage. Je remercie tout particulièrement Monsieur Eveno, mon enseignant tuteur, pour le temps consacré au suivi de mon stage et pour son accessibilité lors de celui-ci. Je remercie également Jean-François Kamp pour sa complète implication et sa disponibilité lors de notre recherche de stage, ainsi que pour l'ensemble des outils de recherche de stage mis à notre disposition.

Je remercie l'ensemble des collaborateurs de Capgemini qui m'ont accueilli lors de mon arrivée dans l'entreprise et en particulier Pierre Foucaud. La procédure d'accueil de stagiaire à Capgemini étant éprouvée, j'ai pu immédiatement trouver ma place dans l'entreprise. La manière dont les stagiaires sont responsabilisés et considérés à Capgemini est unique et je remercie pour cela Capgemini de m'avoir accueilli.

Table des matières

1.	Introduction	1
1.1.	Présentation de l'entreprise	1
1.2.	Présentation d'Aloni, projet encadrant le stage	5
2.	Développement d'un composant d'IHM en arborescence	14
2.1.	Introduction	14
2.2.	Fonctionnalités à implémenter	15
2.3.	Présentation du composant natif de JavaFx	22
2.4.	Création d'un composant personnalisé	24
2.5.	Évolution de la méthode de travail	33
2.6.	Conclusion de la première partie	34
3.	Adaptation du composant d'IHM	35
3.1.	Introduction	35
3.2.	Structure de données du projet Aloni	35
3.3.	Actions et évènements dans le projet Aloni	38
3.4.	Spécification de la surcharge	38
3.5.	Conclusion de la deuxième partie	41
4.	Conclusion	42
5.	Glossaire	43
6.	Table des figures	44

Note : Les mots en *bleu* sont définis dans le Glossaire, p.46

1. Introduction

1.1. Présentation de l'entreprise

1.1.1. Le groupe Capgemini

Capgemini est une entreprise de conseil et services informatiques, plus communément appelée **SSII**. Affichant un effectif total de 180 000 collaborateurs dans plus de 40 pays, Capgemini est l'un des leaders mondiaux sur ses domaines de compétence. Le Groupe Capgemini a réalisé en 2015 un chiffre d'affaires de 11,9 milliards d'euros, ce qui l'a placé à la sixième place du classement mondial des SSII par chiffre d'affaires.

Capgemini distribue ses services en quatre grandes catégories :

- Services de conseil (Capgemini Consulting) : Services d'identification, de développement et de mise en œuvre des projets de transformation dans le but de renforcer la croissance et d'accentuer l'avantage compétitif des entreprises clientes.
- Services d'intégration de systèmes : la conception, le développement et la mise en œuvre de projets technologiques couvrant l'intégration de systèmes complexes et le développement d'applications informatiques.
- Services informatiques de proximité (SOGETI) : Services technologiques professionnels pour répondre aux besoins locaux en termes d'infrastructures, d'applications, d'ingénierie, de test et d'opérations.
- Services d'infogérance : Service d'aide à la gestion des systèmes d'information et des activités connexes. Ces contrats les lient à des entreprises pour une durée allant de cinq à dix ans.

1.1.2. Lieu de travail

J'ai effectué mon stage dans les locaux de la branche Rennaise de Capgemini, au pôle tertiaire « Le Spirea » situé à la frontière entre Rennes et Cesson-Sévigné.



FAÇADE DU SPIREA

Le pôle Rennais de Capgemini emploie environ neuf cent employés, pour la plupart ingénieurs. Ils recrutent en moyenne cent cinquante de nouveaux collaborateurs chaque année, dont environ la moitié au terme d'un stage de fin d'études d'ingénieurs. L'année dernière, seuls quatre des soixante-dix stagiaires ne se sont pas vus offrir de CDI au terme de leur stage.

Ce pôle travaille en collaboration avec des entreprises locales, régionales ou nationales, comme par exemple la SSII Silicom, le pôle [DGA-MI](#) de Brux, ou la chaîne M6.

1.1.3. Responsable au sein de l'entreprise



PIERRE-ANTOINE BONNIN

L'intitulé du poste de Pierre-Antoine est « Architecte Solution S ».

Afin de mieux comprendre ce rôle, je vais vous présenter succinctement l'organisation des tâches dans un projet informatique.

Un projet informatique voit le jour dans le but de produire une [solution technique](#) capable de répondre au mieux au besoin d'un client. Trois principaux acteurs permettent à l'entreprise prestataire en charge du projet de produire la solution adaptée.

Principal interlocuteur avec le client, l'expert technique définit une liste exhaustive de fonctionnalités et de [capacités](#) (hautes performances, compatible des réseaux bas débit...) nécessaires afin de répondre pleinement au besoin exprimé par le client.

Cette liste de fonctionnalités et capacités est ensuite étudiée par l'Architecte Solution S, épaulé si besoin par des assistants architectes de grades moins élevés, afin de proposer une solution technique. Cette solution doit implémenter l'ensemble des exigences définies par l'expert technique, tout ayant la dimension adaptée au besoin du client. Il doit donc définir les technologies utilisées ainsi que l'architecture technique de la solution.

Cette proposition de solution technique est ensuite développée et déployée par les équipes techniques, sous la supervision du Responsable Technique.

Une fois la solution technique développée, et les tests qualités effectués, la solution est prête à être livrée au client. Celui-ci effectuera ses propres tests de qualités, afin d'accepter ou non la livraison.

Le diagramme suivant illustre cette organisation :

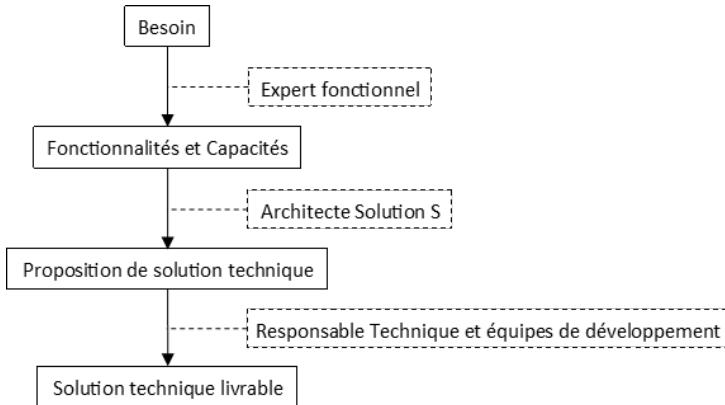


FIGURE 1 - ORGANISATION DANS UN PROJET DE DEVELOPPEMENT INFORMATIQUE

Pierre-Antoine Bonnin est architecte solution S, mais il joue également le rôle d'expert technique dans des projets de petite envergure. Il peut également lui arriver d'être responsable technique de projets si ceux-ci sont de taille suffisamment modeste.

1.2. Présentation d'Aloni, projet encadrant le stage

1.2.1. Contexte

Le projet Aloni est un projet d'étude pour le compte de la Direction Générale de l'Armement, Maîtrise de l'Information.

La DGA-MI déploie de nombreux systèmes informatiques. Chacun de ces **déploiements** met en jeu l'assemblage d'un grand nombre de ressources matérielles (serveur, câbles réseaux...) et logicielles (systèmes d'exploitation, licence applicative...), dans le but d'assurer un niveau de performance et un niveau de service donné à l'échéance prévue. L'ensemble d'un système n'est pas nécessairement déployé en une seule itération. Il est fréquent que les systèmes soient déployés par étapes, ou dans des états de fonctionnement incomplet.

La problématique se pose donc ainsi :

Comment assurer le bon déroulement du déploiement afin que l'ensemble des systèmes déployés soient opérationnels au niveau attendu et à l'échéance prévue ?

Cette problématique se décompose en un certain nombre de problèmes, induits par les caractéristiques de tels déploiements :

- la multiplicité des acteurs, entraînant une difficulté accrue pour identifier l'expert compétent lorsque le besoin s'en fait sentir, connaître ses coordonnées, s'assurer de la compatibilité de son emploi du temps avec le planning de déploiement... ;
- la multiplicité des ressources, causant des difficultés pour assurer la disponibilité de l'ensemble des ressources nécessaires au moment voulu, connaître l'état des stocks et les fournisseurs, connaître l'ensemble des ressources dépendant d'une ressource défaillante... ;
- la sensibilité potentielle des informations, nécessitant d'assurer un accès restreint aux informations et de garder une trace des opérations effectuées ;
- la complexité de l'assemblage des ressources, rendant primordial la possession de l'ensemble des documents regroupant les informations nécessaires au déploiement de cette ressource ainsi que la cohérence de ces documents avec les caractéristiques de cette ressource ;
- etc.

Il apparaît donc nécessaire de disposer d'un outil centralisé et sécurisé, utilisable par tous les acteurs d'un déploiement et permettant de regrouper l'intégralité des informations concernant ce déploiement.

1.2.2. Études préliminaires

Deux contrats d'études ont été effectués pour ce projet avant mon arrivée à Capgemini. J'ai donc dû m'approprier le contenu de ces études pour que mon travail s'inscrive dans leur continuité.

1.2.2.1. État de l'art

La première de ces études consiste en un état de l'art de **COTS** (*Solution sur étagère*) pertinents dans le cadre du besoin exprimé par la DGA-MI. Il est en effet nécessaire de savoir s'il existe sur le marché des solutions répondant au besoin avant d'engager un éventuel contrat de développement.

Cette étude commence par présenter le contexte de l'étude, puis décline la liste des problématiques posées par la préparation et le déroulement d'un déploiement.

Les problématiques sous-jacentes peuvent être déclinées en deux axes :

- **Etablir des processus** : Comment établir des processus de déploiement permettant aux systèmes déployés d'être opérationnels ?
- **Maximiser les performances** : Comment minimiser la durée de ces processus en maximisant les performances finales ?

Pour la préparation et l'exécution d'un déploiement, les problématiques sont déclinées dans la matrice que vous pourrez trouver en Annexe. La problématique initiale correspond à la cellule supérieure gauche, mais le périmètre de l'étude a été élargi afin de ne pas ignorer d'idées qui se seraient avérées pertinentes par la suite.

On passe ensuite en revue les COTS, en analysant leurs qualités et leurs défauts pour déterminer s'il répond correctement au besoin et peut être réutilisé ou adapté. Chaque analyse apporte un certain nombre d'idées et de concepts réutilisable dans la définition d'une solution répondant au besoin.

La conclusion de cette étude n'identifie aucun CTOS répondant au besoin. En effet, aucun logiciel ne s'est avéré suffisamment généraliste, ergonomique ou proche fonctionnellement de la solution attendue pour s'avérer utile en soi. Leur étude a cependant permis de constituer une base de travail pour la seconde étude.

Phase de préparation

PREP01 Pré-requis - Quelles sont les ressources pré-requises au déploiement ? (Ressources matérielles / applicatives / humaines / etc.)	PREP02 Modélisation - Comment modéliser et structurer des pré-requis génériques adaptables à tout type de système déployé ?	PREP03 Inventaire - Quelles sont les ressources disponibles ?
PREP04 Réservation - Comment réserver les ressources nécessaires ?	PREP05 Planification - Comment identifier et ordonner l'usage des ressources ?	PREP06 Alertes - Comment identifier les points d'attention à remonter à l'utilisateur ?
PREP07 Dimensionnement - Comment dimensionner les ressources afin de répondre à une qualité de service attendue ?		

Phase de déploiement

DEPL01 Ordonnancement - Comment suivre les différentes étapes à suivre pour déployer le système ?	DEPL02 Ressources - Comment récupérer les ressources réservées pendant la préparation ?	DEPL03 Avancement - Comment connaître l'état d'avancement du déploiement en cours ?
DEPL04 Experts - Comment identifier et contacter les experts capables d'aider à la résolution des problèmes rencontrés ?		

DEPL10 Génération - Comment capitaliser sur les informations déjà saisies afin d'accélérer la rédaction des pièces de travail / contractuelles requises en amont des déploiements ? (ex. SILRIA)	DEPL11 Capitalisation - Comment capitaliser sur les déploiements déjà effectués afin d'améliorer les suivants ?	DEPL12 Travail collaboratif - Comment permettre le travail collaboratif sur le processus de déploiement de plusieurs utilisateurs pouvant avoir des profils et responsabilités différents ?
PREP10 Import - Comment accélérer le provisionnement des éléments d'entrées systématisques dont le format est connu ?	PREP11 Capitalisation - Comment capitaliser sur les déploiements déjà effectués afin d'améliorer les suivants ?	PREP12 Capitalisation - Comment capitaliser sur les déploiements déjà effectués afin d'améliorer les suivants ?
PREP13 Travail collaboratif - Comment permettre le travail collaboratif sur la préparation au déploiement de plusieurs utilisateurs pouvant avoir des profils et responsabilités différents ?		

Etablir des processus

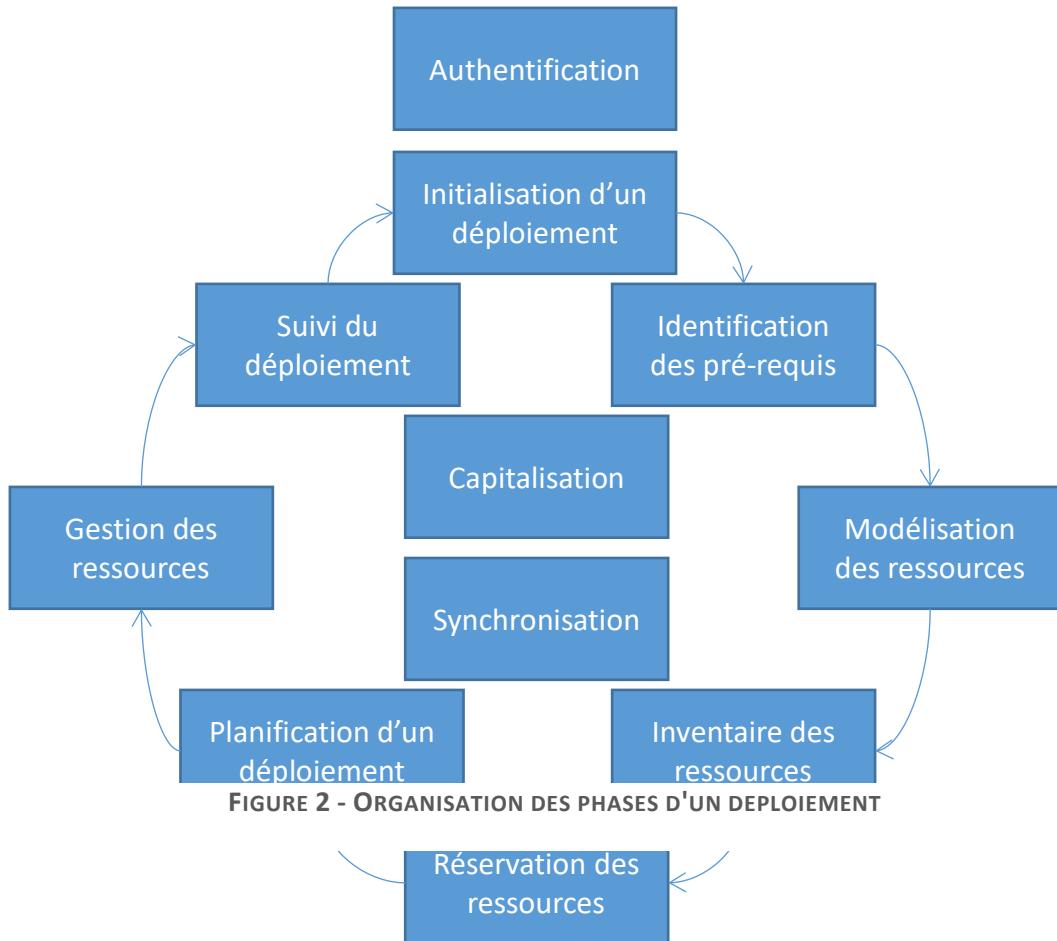
Maximiser les performances

1.2.2.2. Analyse d'une solution complète

La seconde étude réalisée avant mon arrivé est nommée « Analyse d'une solution complète répondant au besoin de validation de la faisabilité des déploiements » et tente de définir de manière exhaustive les caractéristiques d'une telle solution.

Elle comprend un rappel de la problématique, une liste des phases constituant un déploiement et une définition des composants communs de ces phases.

L'analyse des phases d'un déploiement n'est pas l'objet de ce rapport, mais vous pourrez trouver leur énoncé dans le graphique ci-dessous.



La phase d'authentification intervient en tout premier lieu, tandis que les phases de synchronisation et de capitalisation sont transversales à toutes les autres.

La Synchronisation désigne la capacité de la solution à garder une cohérence dans ses données sur tous les postes concernés.

La Capitalisation permet d'accélérer l'utilisation de la solution en proposant une sauvegarde des travaux effectués afin de les adapter et de les réutiliser.

Ce sont les composants communs à ces phases qui concernent plus particulièrement mon travail au sein de ce projet. Vous trouverez ci-dessous l'extrait du document introduisant ces composants communs :

La création d'un outil de vérification de faisabilité d'un déploiement offre un ensemble de vues dépendant de la phase et du rôle de l'utilisateur. Afin de disposer d'un outil cohérent, il est nécessaire de mettre en place des composants communs qui seront à la base des différentes vues.

Une vue schématique des composants communs est la suivante :

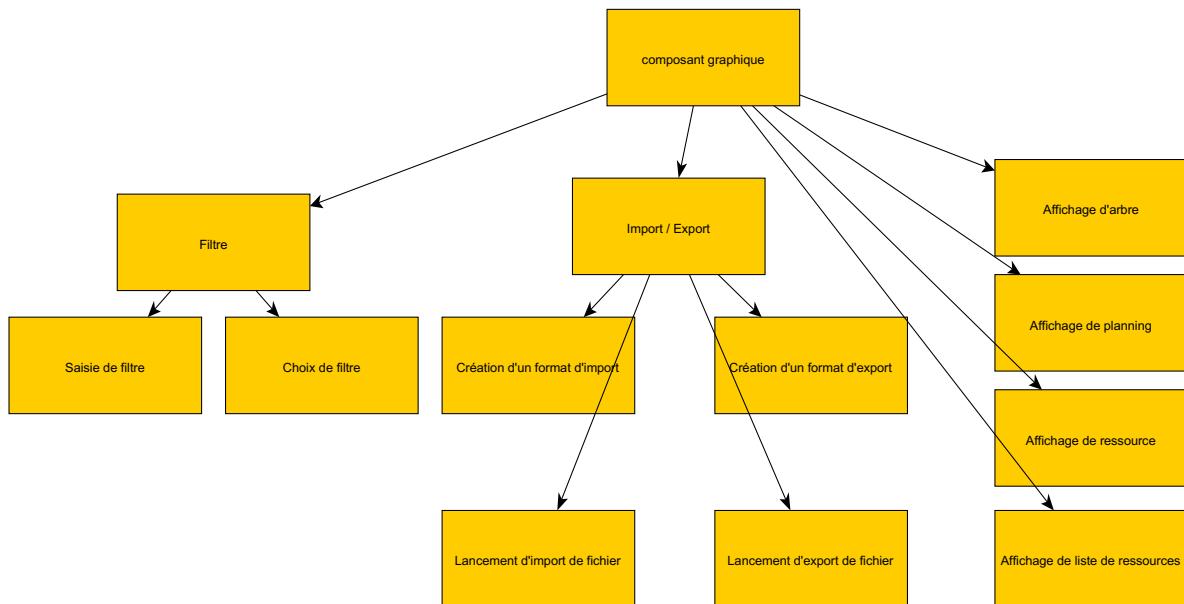


FIGURE 3 - COMPOSANTS COMMUNS AUX DIFFERENTES PHASES D'UN DEPLOIEMENT

J'ai principalement travaillé lors de ce stage sur le composant d'affichage d'arborescence des ressources. Afin de comprendre la nature de mon travail, il est nécessaire de connaître certains principes fondateurs de la solution proposée.

1.2.3. Principes fondamentaux

Ces principes, bien qu'utilisés lors de toutes les phases d'un déploiement, sont principalement liés à la phase de modélisation des ressources, succinctement définie comme suit dans le document :

Phase destinée à un profil d'architectes qui vont définir le modèle de données à manipuler. Cette modélisation peut évoluer avec les besoins de déploiement identifiés.

Afin de préparer efficacement un déploiement, il est nécessaire de pouvoir modéliser et manipuler l'ensemble des données le concernant. Cette phase de modélisation transforme les données en « ressources » manipulable et ordonnable grâce à des « modèles ».

La modélisation de ressources comporte donc deux phases sous-jacentes : la création de modèles de ressources et l'instanciation de ces dites ressources.

Une ressource peut représenter tout type de données. Dans le cadre de la solution proposée, une ressource peut par exemple être matérielle (routeur, serveur, câble...), logicielle (système d'exploitation, licence logiciel...) ou encore humaine (annuaire de personnel, clients...).

Chaque ressource est munie d'attributs. Ces attributs peuvent être de type simple (texte, nombre, date, document informatique...) ou bien contenir une ou plusieurs autres ressources. Ces « sous-ressources » peuvent être liées à leur ressource parente par « possession » ou par « référence ».

Une dépendance de type « possession » signifie que la sous-ressource n'est présente que comme enfant de sa ressource mère. C'est le cas par exemple de la ressource « Carte-réseau » enfant d'une ressource « Serveur ». Une même carte réseau ne peut être dans plusieurs serveurs en même temps.

A l'inverse, une dépendance de type « référence » permet comme son nom l'indique de référencer une ressource extérieure. Une ressource « Serveur » pourra par exemple référencer un « Administrateur » comme responsable, car le même administrateur peut être responsable de plusieurs serveurs.

Pour construire ces ressources, il est nécessaire de créer des modèles de ressource. Chaque ressource peut s'appuyer sur un ou plusieurs modèles de ressources. Cette ressource hérite alors de tous les attributs de ses modèles, et ces attributs prennent comme valeur par défaut la valeur de leur modèle. Modifier un modèle modifie en conséquence toutes les ressources s'appuyant sur lui, à l'exception de celle qui ont vu leurs valeurs modifiées au cas par cas.

Les modèles eux-mêmes peuvent hériter d'autres modèles, soit pour surcharger la valeur de leurs attributs ou pour ajouter des aspects à une ressource. Par exemple le modèle « Serveur à vendre » héritera de manière évidente des caractéristiques du modèle « Serveur », mais également de celles du modèle « Produit commercial ». Ainsi le modèle « Serveur à vendre » pourra être utilisé pour créer l'ensemble des ressources représentant les serveurs en stock qui ont pour but d'être vendu.

Le schéma suivant reprend l'exemple précédent.

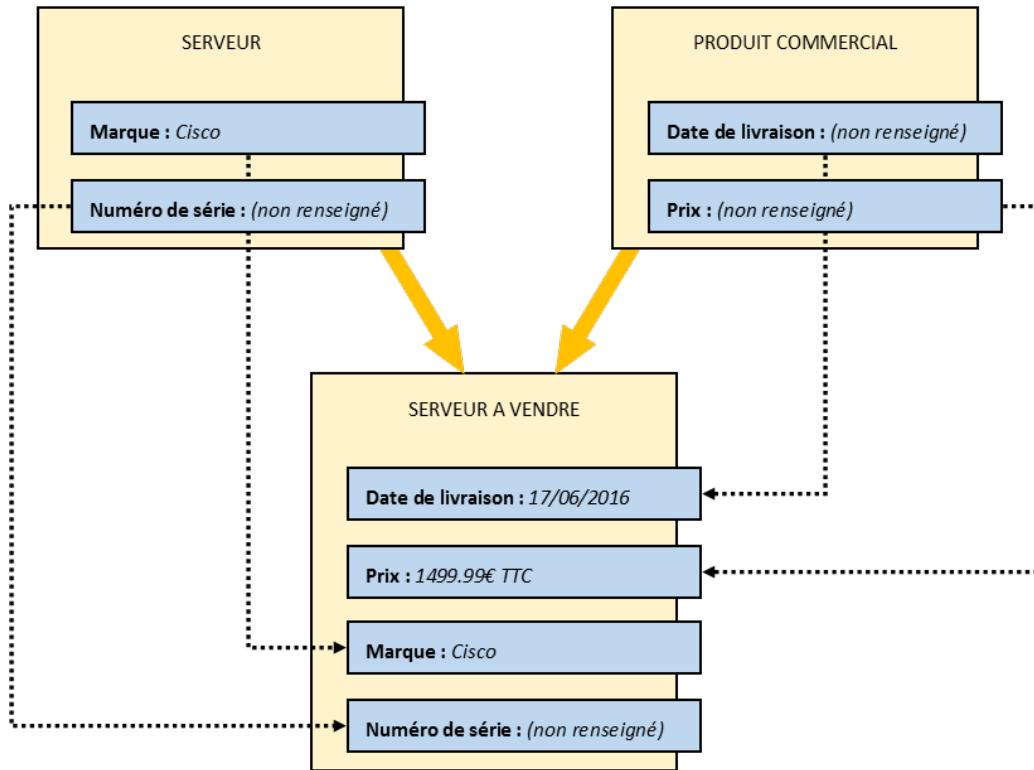


FIGURE 4 - EXEMPLE D'HERITAGE

Cet exemple simple permet de comprendre qu'une ressource (modèle ou non) peut hériter de plusieurs modèles. Cependant chacun des attributs d'une ressource ne peut hériter que d'un seul attribut modèle, comme symbolisé par les flèches en pointillé sur la figure ci-dessus.

En considérant l'ensemble de ces concepts, on comprend que deux arborescences font partie intégrantes de la solution proposée :

- L'arborescence des modèles de ressource, représentant les relations d'héritages liant les modèles et les ressources entre eux ;
- L'arborescence des instances de ressources, de leurs attributs et de leurs « sous-ressources » ;

Pour visualiser et manipuler ces deux arborescences, il est nécessaire de disposer d'un composant d'affichage en arbre modulable et disposant de nombreuses fonctionnalités adaptées.

1.2.4. Présentation du travail attribué

Au terme des deux études précédentes, il a été décidé qu'un prototype exploitant les données des études devrait être réalisé.

Cette notion de « prototype » signifie que le travail effectué n'a pas pour but d'être utilisé directement, mais pourra servir de base à la définition d'un futur projet. Cela permet également de tester la validité d'idées ou propositions esquissées précédemment.

Ces projets offrent une plus grande liberté au prestataire que les projets « classiques », car le but est alors de prouver la validité des idées avancées plutôt que de fournir un produit fonctionnel et complet. Cela permet notamment, lors de la programmation, de se concentrer davantage sur la capacité d'évolution, la stabilité et la maintenabilité du code source plutôt que sur l'ajout d'un nombre précis de fonctionnalités au détriment de sa qualité.

Mon travail a donc été de travailler à la réalisation de ce prototype.

Dans un premier temps, j'ai dû développer un composant d'affichage en arborescence destiné à être réutilisé dans de nombreux projets. Afin d'atteindre ce but, il m'a fallu garder un certain niveau de généralisation dans la définition du composant.

J'ai ensuite dû utiliser le composant graphique précédemment créé pour l'adapter à l'usage prévu dans le cadre de ce projet. J'ai donc dû créer une interface entre mon composant graphique généraliste et le [noyau](#) du projet. Lors de cette partie, j'ai également manipulé le noyau de l'application.

1.2.5. Technologies utilisées



J'ai exclusivement travaillé avec le langage de programmation orienté objet Java, produit par Oracle.

Pour développer, j'ai utilisé l'IDE libre NetBeans, comparable à Eclipse.



Pour créer et manipuler les interfaces graphiques, j'ai utilisé la librairie graphique [JavaFx](#) qui est la librairie graphique de Java recommandée par Oracle.

Cette librairie est compatible avec la librairie Swing, présentée à l'IUT, et de nombreux concepts sont similaires entre ces deux librairies. JavaFx améliore et facilite cependant la création d'[IHM](#) en proposant notamment la création d'interface via des fichiers XML et CSS.

La gestion de code source s'est effectuée grâce à Maven et Git. Maven permet une uniformisation des normes dans les codes sources de projets Java ainsi que les outils nécessaires à leur compilation et leur déploiement.



TortoiseGit
Windows Shell Interface to Git

Git, quant à lui, propose des fonctionnalités de sauvegarde, d'historique et de partage de code source. J'ai utilisé l'interface TortoiseGit qui permet de facilement créer et manipuler des dépôts gits par le biais de l'explorateur Windows.

2. Développement d'un composant d'IHM en arborescence

2.1. Introduction

La plupart des librairies graphiques, JavaFx y compris, incluent leur propre composant d'affichage en arborescence. Bien qu'ils offrent de nombreuses possibilités de personnalisation, ils sont prévus pour un usage simple.

Cela rend la création d'une surcouche indispensable afin de l'utiliser pour des usages complexes tels que celui qui nous intéresse.

Bien que créé en ayant à l'esprit les contraintes du projet Aloni en particulier, le composant graphique en arborescence que j'ai dû définir met en œuvre un certain nombre de concepts pouvant s'avérer utiles à d'autres projets. Le but de mon travail a donc été de créer une [API](#) suffisamment généraliste pour que son usage soit possible dans de nombreux projets, mais à l'usage suffisamment simple pour justifier la création d'une [surcouche](#) sur le composant d'affichage natif de JavaFx.

La création de ce composant graphique, qui sera par la suite appelé explorateur, doit répondre à deux enjeux majeurs.

Il doit tout d'abord supporter un grand nombre de fonctionnalités complexes, tout en offrant une ergonomie forte, adaptée à un usage par différents profils d'utilisateurs. De cet enjeu découle le choix de JavaFx, qui est une technologie possédant à la fois la performance et la facilité de prise en main nécessaires pour ce projet.

Cet explorateur continuera à être utilisé par la suite dans de nombreux contextes, il a donc été nécessaire d'avoir une réelle réflexion autour de son architecture logicielle pour offrir une implémentation progressive de l'ensemble des fonctionnalités attendues. Il a donc fallu consentir à un réel effort initial dans sa conception plutôt que de procéder à une approche naïve de la programmation qui aurait mené à de fréquents retours en arrière sur le travail accompli.

2.2. Fonctionnalités à implémenter

Je ne détaillerais pas dans cette partie l'ensemble des fonctionnalités que j'ai dû implémenter, mais les principales. Je vais présenter dans un premier temps les fonctionnalités ou capacités qui ont nécessité un travail sur la structure de mon code, puis les fonctionnalités indispensables pour répondre au besoin, mais dont l'implémentation n'a pas joué sur cette structure.

2.2.1. Fonctionnalités structurantes

Gestion du nombre important d'éléments

Une des principales contraintes des déploiements est le nombre parfois très important de ressources à manipuler. L'arborescence à générer est alors très importante, et il devient nécessaire de séparer le traitement des données de leur affichage. Les données ne doivent donc pas être intégralement chargées dès l'initialisation de l'explorateur, mais au fur et à mesure du parcours de l'arborescence. Cette considération est particulièrement vraie si les données sont stockées sur une autre machine que celle où l'application s'exécute et qu'il est alors nécessaire de télécharger les données par un réseau dont le débit peut être variable.

De même les fonctionnalités de recherche dans l'arbre doivent être visuellement présentes dans l'explorateur, mais la recherche en elle-même ne doit pas être traitée par le composant graphique. La recherche en elle-même doit être déléguée au composant métier qui implémentera l'explorateur. Cela permettra par exemple de s'appuyer sur les fonctionnalités des bases de données qui sont optimisées pour cet usage précis.

Multiplicité des vues sur un même ensemble de données

Le but de cet explorateur est de proposer un affichage complet, modulable et ergonomique d'une arborescence de données et ce malgré la taille potentielle de l'arborescence.

Pour cela, l'utilisateur doit pouvoir ouvrir plusieurs [vues](#) sur le même ensemble de données. Cela lui permettra par exemple de visualiser plusieurs parties de l'arborescence en même temps, comparer deux branches de l'arbre facilement, faciliter les actions de glissé-déposé... Toute modification sur une vue de l'arbre doit automatiquement et immédiatement être reportée dans toutes les autres vues de l'arbre.

Il doit également être possible de définir un nœud comme racine de travail sur une vue, tout en laissant la possibilité à tout instant de revenir à la racine native de la vue. Cela permet de se concentrer sur une partie précise de l'arbre.

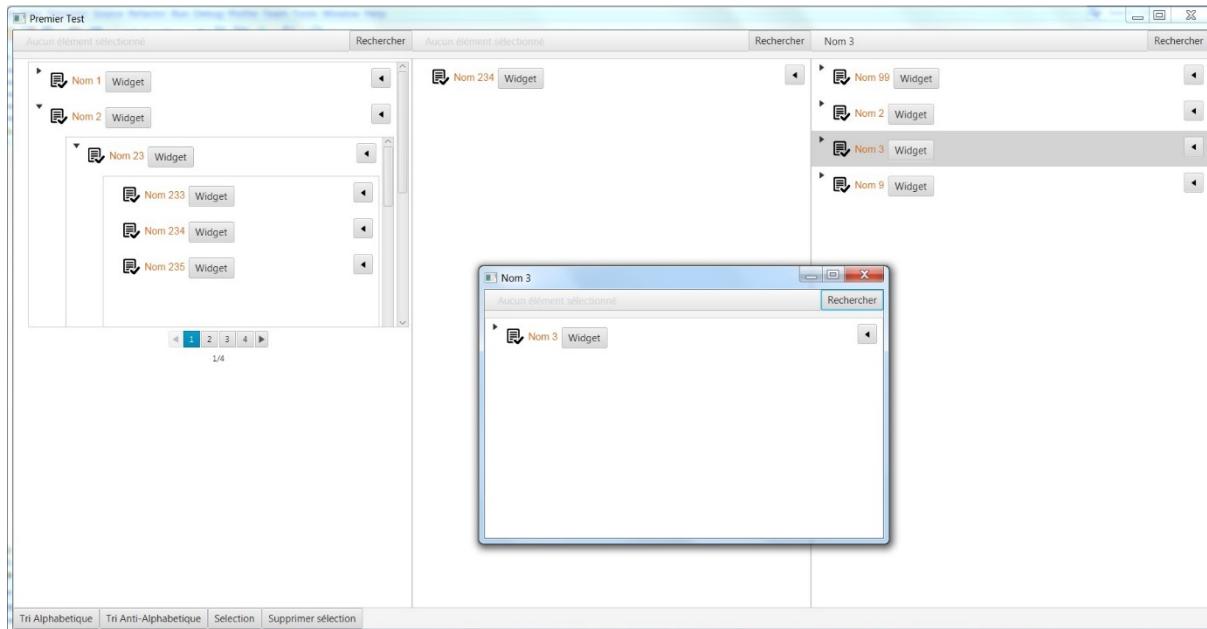


FIGURE 5 - DÉMONSTRATION DE LA MULTIPlicité DES VUES

Dans la capture d'écran ci-dessus, le même ensemble de données est visualisé dans quatre vues différentes, dont une dans une fenêtre séparée. Chacune de ces vues a des racines différentes, mais représente le même jeu de données.

Parents multiples pour un même élément

L'explorateur doit offrir la possibilité de définir plusieurs parents pour un seul élément. Cette contrainte découle notamment de l'héritage multiple dans le cadre du projet Aloni.

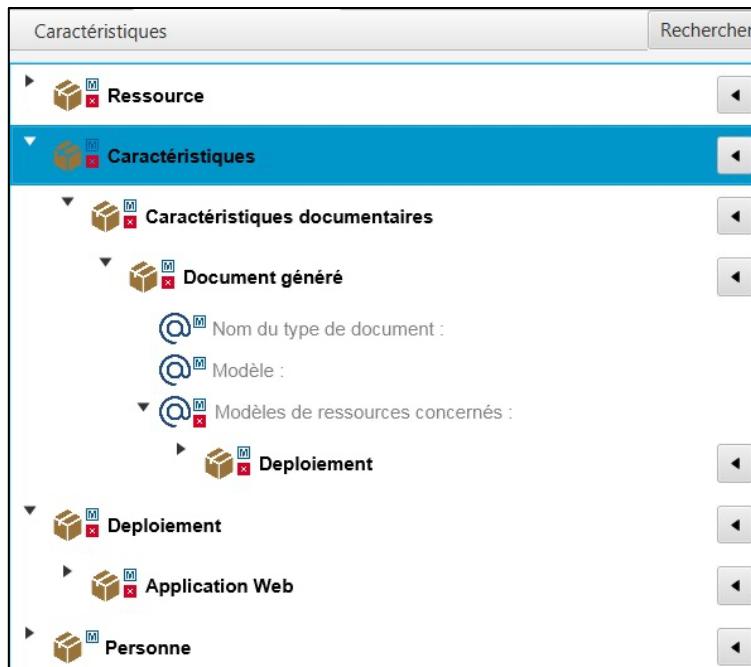


FIGURE 6 - EXEMPLE DE MULTIPlicité DES PARENTS

Dans l'exemple ci-dessus, l'élément « Deployment » est présent deux fois : une fois à la racine de la vue et une fois en tant que sous-élément de « Modèles de ressources concernés ».

La plupart des composants graphiques d'affichage d'arborescence, celui de JavaFx y compris, ne donnent la possibilité que de définir un seul parent. Il a donc fallu réfléchir à un moyen de contourner cette limitation, et cela a influencé la structure de mon code.

Surcharge possible des actions

Suivant l'usage fait de cet explorateur, il se peut que les actions effectuées sur les nœuds de l'arbre changent de nature. Un glissé-déposé n'aura pas le même sens dans toutes les applications, et l'explorateur doit alors proposer la possibilité de surcharger les actions associées au glissé-déposé. Cela est vrai pour l'ensemble des actions présentes dans l'explorateur.

Il doit également être possible pour l'application utilisant l'explorateur d'ajouter ses propres actions à l'explorateur, liées à certains nœuds, et d'en définir le comportement.

2.2.2. Fonctionnalités importantes mais peu structurantes

Personnalisation graphique des éléments

Chaque élément de l'arbre peut être personnalisé afin de correspondre à n'importe quel usage.

Graphiquement, chaque élément de l'arbre se présente comme suit :

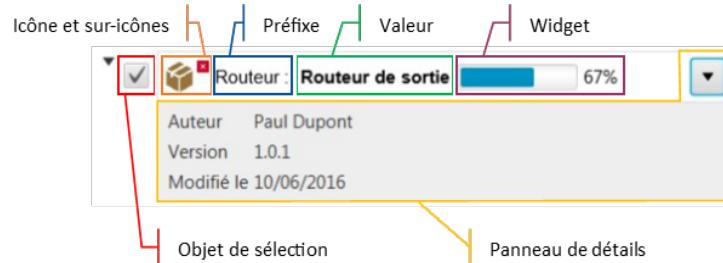


FIGURE 7 - STRUCTURE D'UN ELEMENT DE L'ARBORESCENCE

Ce nœud est volontairement complexe, mais tous ses composants sont facultatifs. Il est donc théoriquement possible de créer des éléments « vides », bien que l'utilité de tels éléments puisse paraître limitée.

Je vais détailler ici le rôle de chacun de ces composants.

Icône et sur-icônes

L'intitulé de chaque nœud peut être personnalisé par une icône représentant la nature de l'élément représenté.

Des « sur-icônes » peuvent être ajoutées pour symboliser une caractéristique supplémentaire. Ces sur-icônes peuvent être contaminants vers la racine ou vers ses enfants.

Pour expliquer l'utilité de ces sur-icônes, je vais prendre l'exemple de l'usage qui en sera fait dans le cadre du projet Aloni.

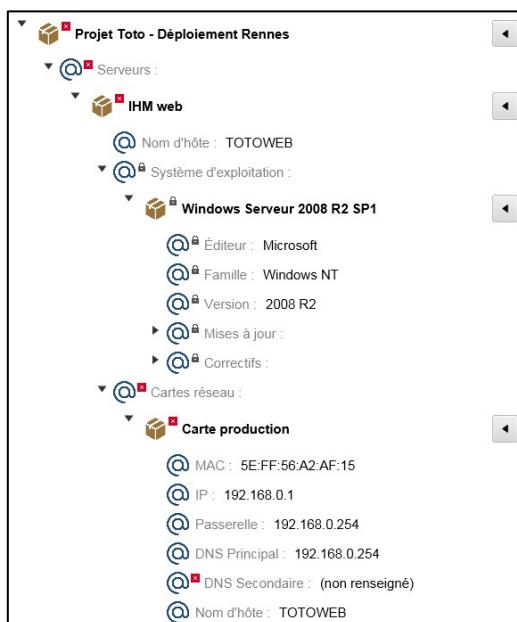


FIGURE 8 - EXEMPLE D'UTILISATION DES SUR-ICONES

En cas d'erreur sur une ressource (ici, l'attribut « DNS Secondaire » n'est pas renseigné), une sur-icône correspondante s'affichera dans l'intitulé de la ressource, mais également dans tous les parents de cet élément jusqu'à la racine. Cela permet de facilement repérer la moindre erreur dans l'arborescence et de trouver son origine en déroulant progressivement les nœuds.

A l'inverse, lorsqu'un responsable verrouille l'édition d'une ressource ou d'un attribut (comme l'attribut « Système d'exploitation » dans l'exemple ci-dessus), cela signifie que tous ses attributs ainsi que les éventuelles sous-ressources sont tous verrouillés également. Dans ce cas, la contamination du sur-icône de verrouillage se fait vers les enfants.

Enfin, pour symboliser qu'une ressource est un modèle, une sur-icône sera utilisée, mais elle ne sera pas contaminante car cette information ne concerne que la ressource elle-même.



FIGURE 9 - EXEMPLE DE SUR-ICONE NON CONTAMINANTE

Préfixe

Ce préfixe nomme la nature de l'élément affiché. La police, la typographie, la taille et la couleur de ce texte peuvent être personnalisé à volonté.

Valeur

Comme le préfixe, son affichage est entièrement personnalisable.

Cependant, sa particularité est de se pouvoir réagir à un double clic pour transformer le texte de manière à modifier la valeur.

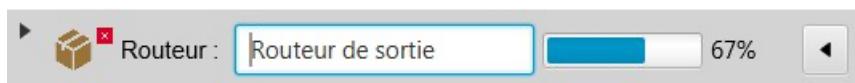


FIGURE 10 - EXEMPLE DE MODIFICATION RAPIDE D'UNE VALEUR

Si par exemple la valeur est de type texte, le double-clic créera un champ de texte. Si la ressource est de type date, un calendrier s'affichera, permettant de sélectionner la date voulue. L'explorateur gère les valeurs de type texte simple, texte multiligne, date, nombre entier et nombre décimal, mais de nouveaux types de valeurs et leur composant de modification peuvent être ajoutés.

Widget

Un emplacement a été prévu pour afficher n'importe quel composant graphique. Une totale liberté d'action est laissée pour cet emplacement, et le composant graphique peut être de n'importe quelle nature (bouton, image, champ de texte, barre de progression...).

Item de sélection

L'explorateur offre la possibilité d'activer la sélection de ses éléments par des cases à cocher ou des boutons radio. La liste des éléments sélectionnés est ensuite récupérable pour effectuer l'action voulue.

Les boutons radio sont toujours utilisés en groupe (donc deux boutons radio au minimum) puisque leur objectif est de permettre à l'utilisateur de choisir une, et une seule, option parmi plusieurs possibles. Les éléments peuvent être répartis en groupe pour permettre une sélection de plusieurs éléments parmi plusieurs groupes.

Panneau de détails

Une liste de détails peut être ajoutée à chaque élément. Cette liste peut être masquée ou affichée à l'aide du bouton fléché à la droite de l'élément.

Menus contextuels et barre de menu

A chaque élément peut être associé des actions qui pourront être trouvées dans le menu contextuel associé à l'élément ou dans la barre de menu lorsque l'élément est sélectionné.

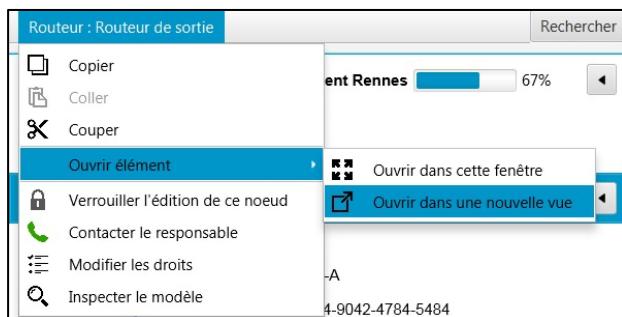


FIGURE 11 - ACTIONS DANS LA BARRE DE MENU

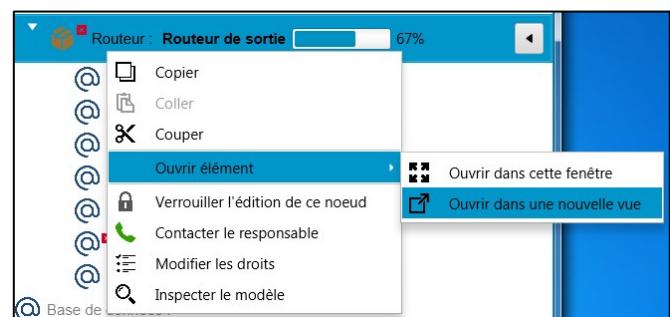


FIGURE 12 - ACTIONS DANS LE MENU CONTEXTUEL

Nœuds « liens »



FIGURE 13 - EXEMPLE DE NOEUD "LIEN"

Des éléments de type « lien » peuvent être ajoutés dans l'arborescence. Un double clic sur cet élément sélectionnera l'élément extérieur référencé. Ces liens seront notamment utilisés pour symboliser les dépendances de type « référence » dans le cadre du projet Aloni.

Pagination

Il est possible de définir combien de sous-éléments un élément doit afficher au maximum. Si cette limite est dépassée, une pagination apparaît pour garder un affichage lisible de l'ensemble des données.

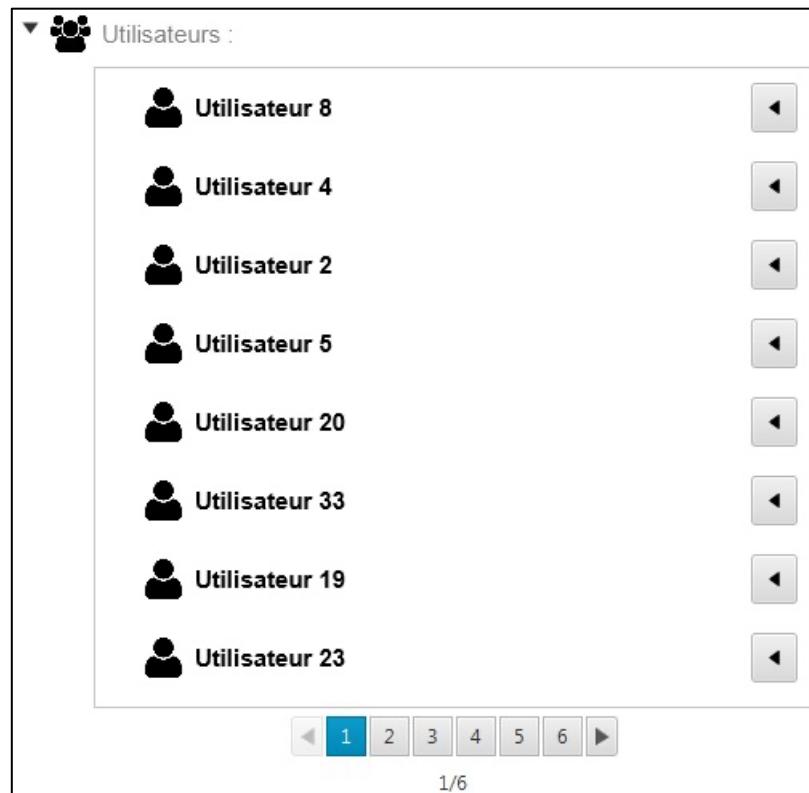


FIGURE 14 - EXEMPLE D'UTILISATION DE LA PAGINATION

2.3. Présentation du composant natif de JavaFx

Dans cette partie, je vais présenter le composant d'affichage d'arborescence natif à JavaFx. Je présenterais les concepts et les terminologies de manière à exposer les adaptations à effectuer.

Voici le type de rendu habituellement obtenu avec le composant d'arborescence proposé par JavaFx.

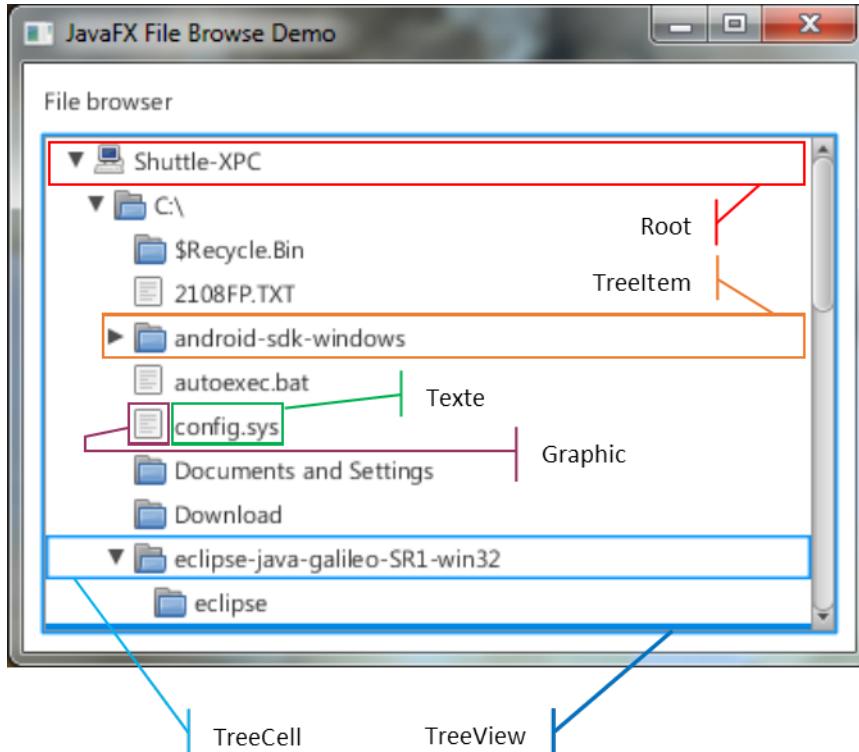


FIGURE 15 - AFFICHAGE D'ARBORESCENCE DE JAVAFX

Le composant d'affichage de l'arborescence est appelé « TreeView ».

Les éléments de l'arborescence sont appelés « TreeItem ». Chaque TreeItem possède un TreeItem parent et peut posséder un ou plusieurs TreeItem enfants. Si un TreeItem possède un ou plusieurs enfants, une flèche de déroulement apparaît à sa gauche.

Une TreeView possède un et un seul TreeItem racine appelé « Root ». La TreeView permet cependant de désactiver l'affichage de sa racine pour simuler la présence de plusieurs racines.

L'affichage d'un TreeItem est décomposé en deux parties :

- Le « Graphic », qui peut contenir n'importe quel élément graphique de la librairie JavaFx.
- Le « Text » qui contient, comme son nom l'indique, un texte fixe.

L'affichage d'un « TreeView » est en réalité décomposé en cellules nommées « TreeCell ». Ce sont ces cellules qui sont en charge de l'affichage des TreeItem qu'elles contiennent ainsi que de leur mise à jour. Leur fonctionnement est trop complexe et trop peu pertinent pour être détaillé dans ce rapport. Il est cependant intéressant de savoir que ce sont ces éléments « cachés » qui seront manipulés pour gérer tout ce qui a trait à la mise à jour d'un TreeItem et ses réactions aux événements graphiques.

Ce composant graphique d'affichage en arborescence présente certains avantages. Il est en effet très ais  de cr er une arborescence simple comme celle pr sent e ci-dessus. Il suffit pour cela de cr er une TreeView, lui attribuer un TreeItem racine et ajouter des enfants   cette racine jusqu'  obtenir l'arborescence d sir e.

De plus, ce composant est pr vu pour  tre personnalis    volont , bien que cette personnalisation puisse parfois  tre relativement complexe. Le fait que le « Graphic » de chaque TreeItem accepte n'importe quel  l ment graphique m'a permis d'y int grer un panneau personnalis  comprenant tous les composants  num r s dans le paragraphe 2.2.2.

2.4. Création d'un composant personnalisé

Dans cette partie, je parlerais des principaux choix structurels que j'ai dû faire dans la création de cet explorateur. Je ne parlerais pas ici de tous les aspects, mais seulement des points les plus importants.

J'utiliserais à partir de maintenant des diagrammes « Pseudo-UML ». Toutes les normes des diagrammes UML ne seront pas respectées au profit d'une plus grande lisibilité.

2.4.1. Définition de l'API

L'API doit être la seule partie du code que le programmeur souhaitant implémenter l'explorateur dans son application aura à manipuler. C'est la partie visible de mon code et doit donc être à la fois complète et lisible pour permettre à n'importe quel programmeur d'utiliser le plus simplement possible l'ensemble des fonctionnalités de l'explorateur.

2.4.1.1. Fonctionnement de la structure de données

L'explorateur est avant toute autre chose une manière d'afficher et de manipuler un ensemble de données organisées en arborescence. Pour cela, la base du fonctionnement de mon API tient en trois classes : l'Explorateur, les Vues Explorateur et les Items.

Le schéma UML suivant est volontairement simplifié pour expliquer les relations liant ces trois classes.

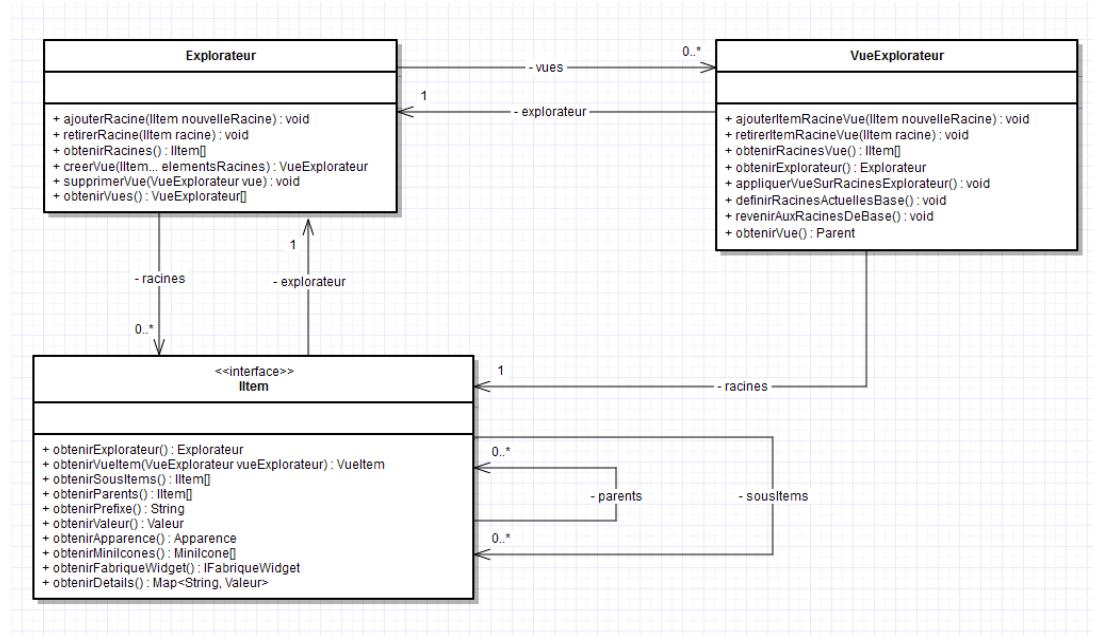


FIGURE 16 - ORGANISATION DU NOYAU DE L'EXPLORATEUR

Le point d'entrée de l'API est la classe Explorateur.

C'est cette classe qui contient les Items racines de l'arborescence de données, et qui crée les vues sur demande.

Les éléments de l'arborescence de donnée sont appelés Items. Ils sont créés à partir de l'interface « IItem » ci-dessus pour être manipulés par le reste du code. Le code de l'explorateur est conçu pour ne manipuler les Items qu'à partir de cette interface. Cela permet à l'utilisateur de l'explorateur de créer ses propres classes implémentant l'interface « IItem ». Ainsi le parcours de l'arborescence est

délégué aux méthodes « obtenirSousItems() » et « obtenirParents() » qui sont définies par l'utilisateur de l'explorateur. Il lui appartient de leur définir le résultat attendu pour assurer une séparation entre le noyau de son application et l'affichage des données.

L'interface Item gère également tout le contenu graphique des éléments de l'arborescence grâce aux méthodes associées.

Associées à un seul explorateur, une multitude de vue peut être créée grâce à la méthode « creerVue(IItem... racines) ». La signature de cette méthode permet de créer une vue avec les Items voulus en tant que racine. Les points de suspension permettent de passer en paramètre un nombre quelconque de IItem. Ainsi, si aucun IItem n'est passé en paramètre, la vue prendra pour racine les racines définies dans l'explorateur. Si au contraire des racines sont précisées, ces IItem deviendront les bases de l'arborescence dans cette vue. Pour intégrer ces vues à l'application utilisant l'explorateur, il suffit d'intégrer à l'interface graphique le « Node » retourné par la méthode « obtenirVue() » de la VueExplorateur. Le « Node » est l'objet graphique basique de la librairie JavaFX, et peut être intégré à toute conteneur JavaFx.

2.4.1.2. Actions, évènements et recherche

L'explorateur est destiné à n'être qu'un affichage des données fournie par le noyau de l'application. Chaque action effectuée sur l'explorateur n'est pas traitée par le code de l'explorateur mais déléguée au noyau. C'est également le noyau qui signale tout changement effectué sur les éléments visualisés pour mettre à jour leur affichage, et qui retourne gère les fonctions de recherche.

Pour mieux comprendre comment ces considérations ont été prises en compte, vous trouverez ci-dessous une description complète de l'interface « IExplorateur », implémentée par la classe « Explorateur ».

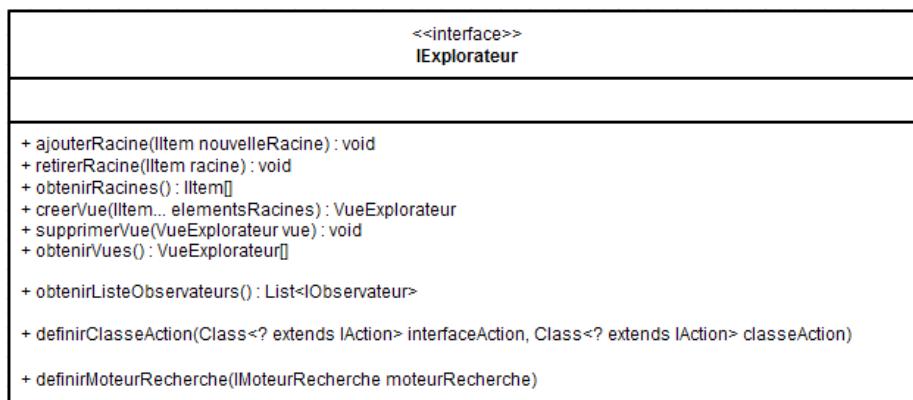


FIGURE 17 - INTERFACE IEXPLORATEUR

Outre les méthodes gérant les Items et les VueExplorateur, trois nouvelles méthodes ont été rajoutées afin de gérer respectivement les observateurs, les actions et la recherche.

Observateurs

La méthode « obtenirListeObservateurs() » retourne un objet de type « List » contenant des objets implémentant l'interface « IObservable ». Le fait de retourner un objet de type « List » permet à la classe appelante de pouvoir parcourir cette liste et y ajouter ou supprimer les « IObservable » à volonté.

L'interface « IObservable » est définie comme suit :

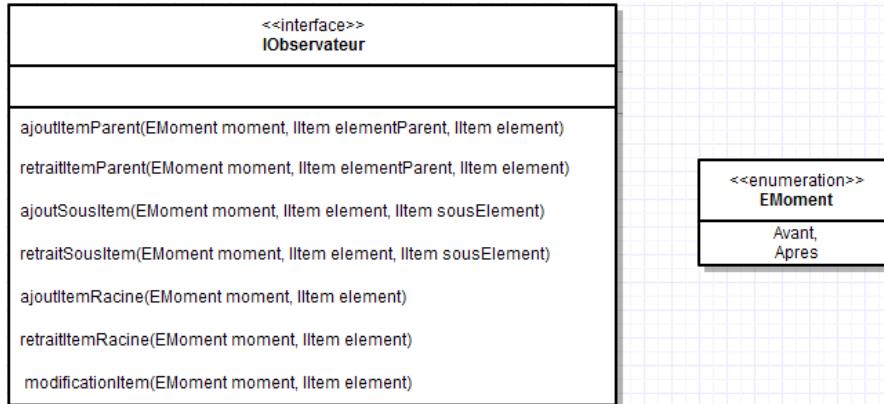


FIGURE 18 - INTERFACE IOBSERVATEUR

Chaque observateur réagit à sept évènements :

- L'ajout d'un IItem parent à un autre IItem ;
- Le retrait d'un IItem parent d'un autre IItem ;
- L'ajout d'un IItem enfant à un autre IItem ;
- Le retrait d'un IItem parent d'un autre IItem ;
- L'ajout d'un IItem racine à l'explorateur ;
- Le retrait d'un IItem racine à l'explorateur ;
- La modification d'un des paramètres (valeur, apparence, détails...) d'un IItem ;

Ce sont les évènements nécessaires pour assurer une mise à jour de l'affichage. Les classes internes de l'explorateur ajoutent ainsi à la liste leur propre instance de « IObservable », en définissant dans chacune des méthodes le comportement qu'elles souhaitent voir s'effectuer à l'exécution d'un des évènements. L'utilisateur de l'explorateur peut également ajouter ses propres observateurs pour faire réagir son application à ces évènements.

Le paramètre « moment » dans chacune des méthodes permet de préciser si l'observateur est notifié avant ou après le déroulement l'évènement, et donc de réagir de manière adéquate.

Lorsque l'un des évènements se produit, il est attendu que l'utilisateur de l'explorateur parcoure la liste des observateurs et appelle la méthode adéquate sur chacun d'eux. Cela demande plus de travail du côté de l'utilisateur mais cela permet de séparer le traitement des actions de leur affichage. Cela demande cependant une gestion des observateurs et des actions pour atteindre ce résultat. Le fonctionnement de ce mécanisme sera détaillé dans le paragraphe suivant traitant des actions.

Actions

Dans le code de l'explorateur, les actions implémentent toutes l'interface `IAction`.

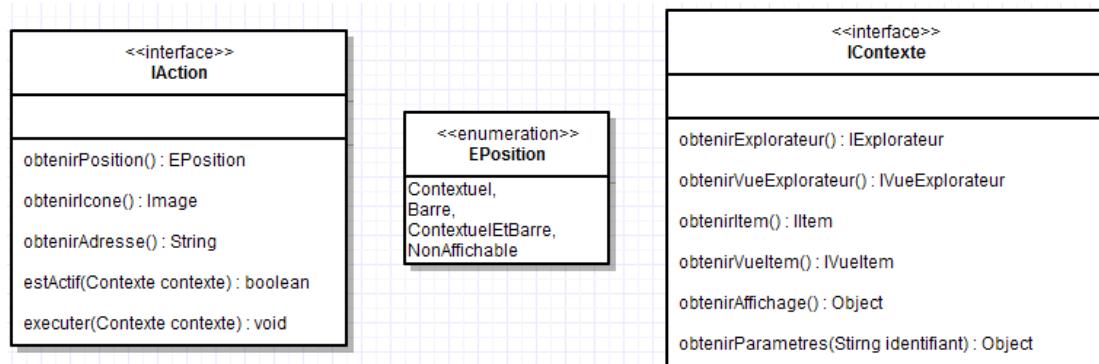


FIGURE 19 - INTERFACES `IACTION` ET `ICONTEXTE`

Les actions peuvent être appelées de deux manières différentes. Premièrement, chaque `Item` peut avoir sa propre liste d'actions qui peuvent être appelées par le menu contextuel ou la barre de menu. Deuxièmement, certaines actions sont appelées automatiquement lors de la manipulation de l'interface. Ces actions sont par exemple celles liées à la modification d'une valeur ou à un glissé-déposé.

L'interface `IContexte` est là pour que l'action puisse connaître le contexte d'exécution de l'action : quel élément l'a appelée, sur quelle vue, etc.

Comme expliqué précédemment, ces actions doivent pouvoir être surchargées par les actions du choix de l'utilisateur. Cela permet à l'explorateur de garder son rôle de simple affichage de données.

Prenons l'exemple de la modification d'une valeur par double clic sur son intitulé. Une fois la nouvelle valeur entrée, l'action associée à la modification d'une valeur est appelée. Cependant, cette action est gérée par le noyau de l'application, qui peut par exemple choisir de ne pas accepter la nouvelle valeur. Si la valeur est cependant bel et bien modifiée au niveau du noyau de l'application, les observateurs présentés dans le paragraphe précédent verront leur méthode « `modificationItem()` » appelée. Alors seulement les classes de l'explorateur auxquels sont reliés les observateurs réagiront pour actualiser l'affichage avec la nouvelle valeur.

Ce mécanisme permet donc à l'application implementant l'explorateur de garder le contrôle sur ses données en appliquant les vérifications nécessaires sur les actions qu'elle définit. Pour définir quelle action appeler, l'application utilise la méthode « **`definirClasseAction(Class<? extends IAction> interfaceAction, Class<? extends IAction> classeAction)`** ». Cette méthode permet d'associer à une interface héritant de `IAction` une action concrète. Ainsi se forme un dictionnaire dans lequel les classes de l'explorateur viennent chercher l'action concrète correspondant aux interactions de l'utilisateur.

Ainsi, un `Item` ne possède pas réellement une liste d'actions, mais une liste d'interfaces servant de clés pour trouver les actions concrètes qui leur sont associées.

Certaines interfaces correspondant aux actions de base sont définies nativement dans le code de l'explorateur, mais le programmeur souhaitant implémenter de nouvelles actions peut créer ses propres actions et les associer aux éléments de l'arborescence.

Recherche

L'explorateur dispose d'un panneau de recherche déroulant permettant de filtrer les éléments affichés dans la vue.

La recherche fonctionne sur la même philosophie que les actions. Le code de l'explorateur manipule les interfaces « IMoteurRecherche » et « IRecherche », tandis que c'est à l'application implémentant l'explorateur de définir le comportement des classes correspondante.

Pour cela, il faut indiquer quel moteur de recherche doit utiliser l'explorateur grâce à la méthode « definirMoteurRecherche(IMoteurRecherche moteurRecherche) ».

Les interfaces utilisées pour la recherche sont définies ainsi :

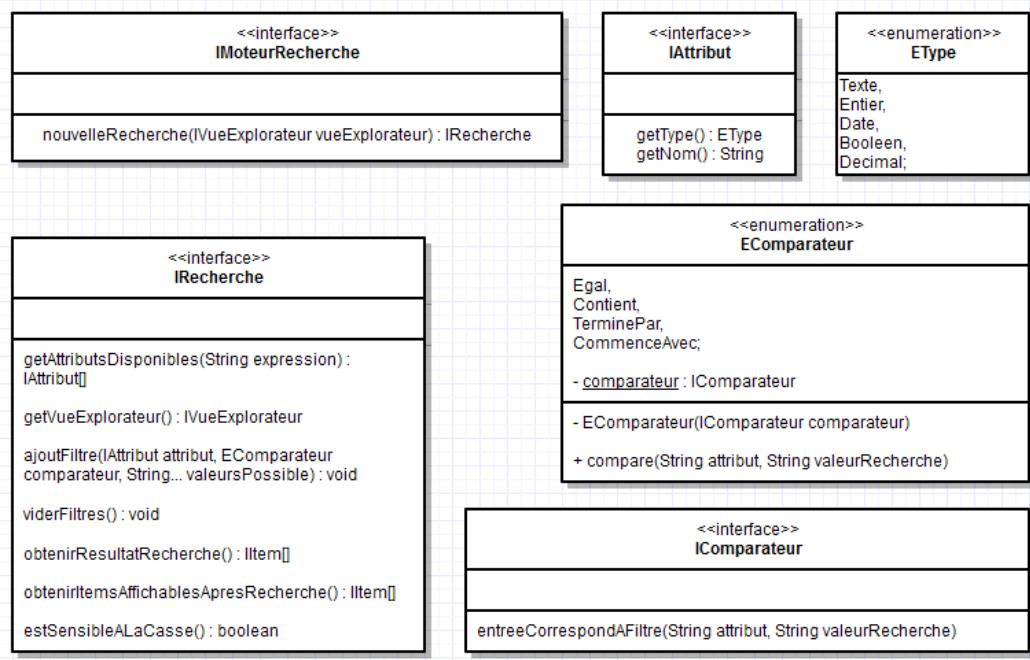


FIGURE 20 - INTERFACES UTILISEES POUR DEFINIR UNE RECHERCHE

Je vais détailler le fonctionnement du panneau de recherche afin de mieux comprendre le fonctionnement de la recherche.

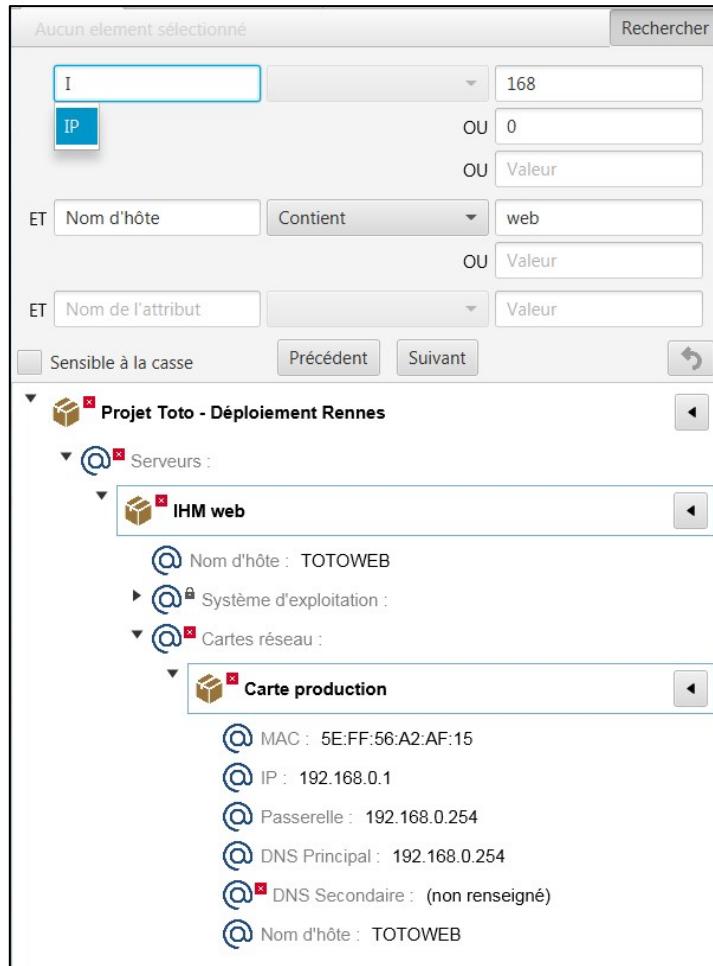


FIGURE 21 - PRÉSENTATION DU PANNEAU DE RECHERCHE

Lorsque le bouton « Rechercher » est enclenché, un panneau de Recherche apparaît, ne comprenant au départ qu'une ligne composée de trois champ : un champ de texte « Nom de l'attribut », une liste déroulante et un champ de texte « Valeur ».

Une case à cocher permet de signifier si la recherche doit prendre la casse en compte, et un bouton permet de réinitialiser la recherche.

Le champ « Nom de l'attribut » est muni d'une auto complétion qui fonctionne grâce à la méthode « **getAttributsDisponibles(String expression) : IAttribut[]** ». Cette méthode retourne la liste des attributs dont le nom commence par le contenu du champ « Nom de l'attribut ». Dans l'exemple ci-dessus, le champ contient la lettre « I », l'auto complétion propose alors le seul attribut dont le nom commence par I, c'est-à-dire « IP ».

Le panneau de recherche se remplit dynamiquement de nouveaux champs au fur et à mesure que l'on remplit les champs encore vides. Si un champ « Nom de l'attribut » et au moins un des champs « Valeur » associés sont remplis, un filtre est ajouté à la recherche grâce à la méthode « **ajoutFiltre(IAttribut attribut, EComparateur comparateur, String... valeursPossibles) : void** » avec

comme paramètres l'Attribut dont le nom est renseigné, le Comparateur sélectionné dans la liste déroulante et les Valeurs renseignées.

La recherche s'effectue alors automatiquement, et deux listes d'Items sont utilisés pour actualiser l'affichage.

La première liste, récupérée par la méthode « **obtenirResultatRecherche() : IItem[]** », retourne la liste des éléments répondant aux filtres de recherche. Dans l'exemple ci-dessus, le seul filtre de recherche complet est le deuxième. Il peut être exprimé ainsi : « L'attribut "Nom d'Hote" contient "web" ». Les éléments encadrés sont donc les Items répondant à ces critères. On peut parcourir les éléments trouvés avec les boutons « Suivant » et « Précédent ».

La deuxième liste peut être récupérée avec la méthode « **obtenirItemsAffichablesApresRecherche() : IItem[]** ». Comme sa signature l'indique, cette méthode permet de récupérer les éléments pertinents après la recherche. Dans l'exemple ci-dessus, les sous-éléments de « Carte Production » ne font pas partie des résultats de la recherche, mais sont tout de même intéressants à afficher car ce sont les attributs de la ressource « Carte Production ».

Il faut garder à l'esprit que la recherche se comporte ainsi car l'application dans laquelle se trouve l'explorateur a défini le moteur de recherche ainsi. Ce comportement pourrait être radicalement différent dans une autre application, ce qui correspond exactement au résultat attendu.

2.4.2. Mécanismes internes de l'API

Dans cette partie, j'exposerais les choix réalisés dans le noyau de l'explorateur. Ce code n'est pas destiné à être manipulé ni même connu d'un programmeur voulant implémenter l'explorateur dans sa solution. Cependant, il m'a fallu définir une structure claire de développement afin de créer un code maintenable et facilement extensible.

2.4.2.1. Plusieurs instances d'un même Item dans une même Vue

Pour répondre à cette contrainte, j'ai dû créer une classe « Vueltem » faisant le lien entre un « IItem » et une « IVueltem ». Cette structure est représentée dans la figure suivante.

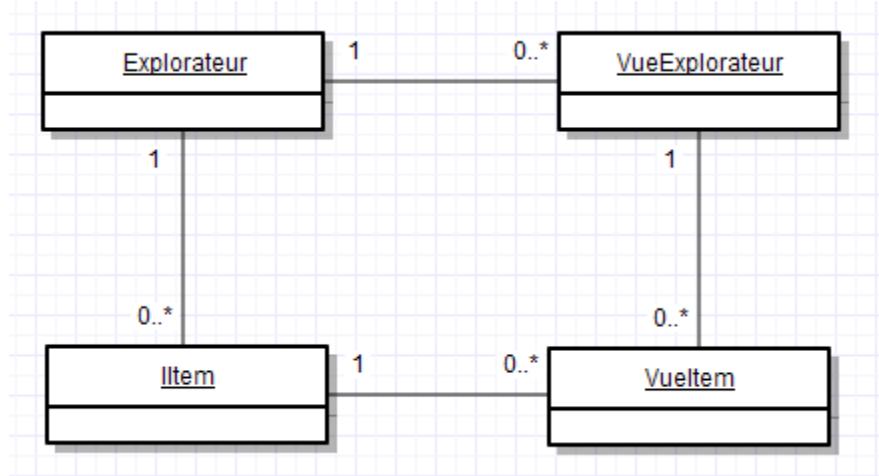


FIGURE 22 - RELATIONS ENTRE LES QUATRE CLASSES PRINCIPALES DU NOYAU DE L'EXPLORATEUR

Afin de comprendre son fonctionnement, il est nécessaire de se souvenir du fonctionnement d'une « TreeView » en JavaFx. L'arborescence d'une « TreeView » est créée à partir de « TreeItem » ayant au maximum un parent et pouvant avoir plusieurs enfants.

Une « Vueltem » possèdera donc un dictionnaire de chacun des « TreeItem » représentant son IItem, associé avec une occurrence d'un parent. Cette structure est nécessaire car chacun des parents du IItem peut également être représenté plusieurs fois. Ainsi même un IItem n'ayant qu'un seul parent pourra être présent plusieurs fois dans la vue si ce parent est lui-même présent plusieurs fois. On associe donc un TreeItem à chacun des TreeItem représentant un parent.

A la mise à jour de l'affichage d'un IItem, c'est donc le Vueltem de chaque vue qui se charge de mettre à jour chacune des occurrences du IItem.

2.4.2.2. Fonctionnement par aspects

Une fois ma structure mise en place correctement, j'ai commencé à ajouter les fonctionnalités une à une : la pagination, les liens, les menus...

J'ai rapidement atteint un stade où la moindre modification entraînait des changements dans la plupart de mes classes et où le moindre bug était plongé dans la masse de différentes fonctionnalités entremêlées.

J'ai donc changé de philosophie de programmation en adoptant la programmation par aspect, que Pierre-Antoine Bonnin m'a présenté et conseillé.

Mes quatre classes principales (Explorateur, VueExplorateur, Vueltem et Iltem) devenaient de simples réceptacles de données, avec leurs accesseurs et modificateurs nécessaires, tandis que chacune des fonctionnalités prenait sa place dans une classe séparée. Ainsi, il suffit d'initialiser chacun des aspects dans ma classe VueExplorateur pour les fonctionnalités se mettent en place. Cela permet d'isoler les problèmes éventuels et d'accélérer grandement l'ajout de nouvelles fonctionnalités.

```

aspectRecherche = new AspectRecherche(explorateur, this);
aspectPagination = new AspectPagination();
aspectTri = new AspectTri();
aspectGlisserDeposer = new AspectGlisserDeposer();
aspectSelectionnerElement = new AspectSelectionElement(aspectPagination);
aspectVueItemsElement = new AspectVueItemsElement(aspectGlisserDeposer);
aspectExtensionItem = new AspectExtensionItem(aspectTri);
aspectCopierColler = new AspectCopierColler();
aspectMenuBarre = new AspectMenuBarre(aspectCopierColler);
aspectMenuContextuel = new AspectMenuContextuel(aspectCopierColler);
aspectReactionClicItem = new AspectReactionClicItem(aspectMenuContextuel);

vueFx = new VueFxExplorateur(this, aspectReactionClicItem, aspectRecherche);

aspects = new AspectsVue();
aspects.getAspects().add(aspectRecherche);
aspects.getAspects().add(aspectSelectionnerElement);
aspects.getAspects().add(aspectPagination);
aspects.getAspects().add(aspectMenuBarre);
aspects.getAspects().add(aspectMenuContextuel);
aspects.getAspects().add(aspectTri);
aspects.getAspects().add(aspectGlisserDeposer);
aspects.getAspects().add(aspectReactionClicItem);
aspects.getAspects().add(aspectVueItemsElement);
aspects.getAspects().add(aspectExtensionItem);
aspects.getAspects().add(aspectCopierColler);
aspects.initialisation(this);

```

FIGURE 23 - SEQUENCE D'INITIALISATION DES ASPECTS

Dans la capture ci-dessus, on voit l'appel des différents aspects dans le [constructeur](#) de ma classe VueExplorateur.

2.5. Évolution de la méthode de travail

Lors de mon arrivé dans l'entreprise, Pierre-Antoine Bonnin était absent pour une semaine de congé. J'ai cependant été très bien encadré par l'équipe d'accueil, notamment Pierre Foucaud, familier avec le projet Aloni, qui m'en a présenté les bases.

J'ai donc commencé par me familiariser avec le projet en lisant les deux études réalisées et le document préparé pour mon arrivée par Pierre-Antoine Bonnin. Je me suis ensuite formé aux concepts fondateurs de JavaFx et aux bases de son utilisation.

Une fois familier avec le fonctionnement de JavaFx, je me suis concentré sur les fonctionnalités à implémenter. J'ai d'abord recherché des solutions techniques et des idées pour répondre aux différentes problématiques se présentant à moi, sans réellement réfléchir à la structure de mon code.

Ce n'est qu'une fois ayant une idée de la manière donc j'allais répondre aux différents problèmes posés par l'utilisation de JavaFx que j'ai commencé à définir la structure de l'explorateur.

Je n'ai pas adopté immédiatement la bonne approche concernant la définition de mon composant graphique : j'ai d'abord défini un premier jet de mon explorateur, qui mêlait les considérations techniques de l'implémentation et les méthodes nécessaires à un utilisateur de cette librairie. L'API était donc polluée par de nombreuses méthodes qui nuisaient à la compréhension de son fonctionnement car inutiles pour un programmeur souhaitant simplement utiliser mon composant.

A son retour de congé, Pierre-Antoine a donc pris le temps de m'expliquer le fonctionnement d'une librairie.

Une librairie se sépare en deux parties : une API et une implémentation. L'API est la façade ne présentant que les composants essentiels à l'utilisation de la librairie tandis que l'implémentation n'est pas destinée à être vue par le programmeur extérieur et contient le code réalisant réellement le travail.

Idéalement, un seul point d'entrée doit être défini dans la librairie. C'est-à-dire qu'une seule classe de l'API pourrait être créée avec l'usage d'un « new ». Cela permet d'isoler totalement le travail effectué par la librairie et le travail effectué par le reste de l'application.

Afin de définir une librairie facile d'utilisation, il est donc nécessaire de définir entièrement l'API, sans se préoccuper de son implémentation. Une fois l'API suffisamment « mature », on peut se concentrer sur la programmation de son implémentation.

Cependant, ce n'est pas parce que l'implémentation n'est pas destinée à être lue par un programmeur extérieur qu'elle ne doit pas respecter une certaine structure. Cette structure doit toujours garder en tête certains principes afin d'être ouvert à l'extension en nécessitant le moins de modification possible.

Pour faciliter l'application de ces principes, on peut s'appuyer sur les « design pattern ».

Dans la programmation objet, les « design pattern » sont des réponses standard à des problèmes récurrents. Ce sont des structures de code dont on peut s'inspirer pour trouver une réponse adaptée aux différents problèmes que l'on peut rencontrer. Mon code utilise donc de nombreux patterns pour différents usages, certains que j'ai appris à l'IUT, d'autres que j'ai découverts pendant ce stage.

2.6. Conclusion de la première partie

J'ai donc défini une librairie permettant de personnaliser à loisir un explorateur en arborescence, doté d'une recherche et d'une collection d'actions pour chaque élément. Son usage peut cependant être fastidieux si l'on doit redéfinir à chaque utilisation l'ensemble des paramètres. C'est pourquoi j'ai créé des classes implémentant les interfaces, facilitant l'usage de l'explorateur en le dotant d'un comportement par défaut.

Ainsi il suffit à l'utilisateur de redéfinir uniquement les actions ou classes qu'il souhaite pour profiter des comportements par défaut que j'ai défini.

Cet explorateur répond à l'ensemble des fonctionnalités demandées. Sa création m'aura cependant demandé sept semaines et a constitué la majeure partie de mon travail. J'ai cependant beaucoup appris lors de ces sept semaines, notamment en ce qui concerne les structures d'architecture de code.

3. Adaptation du composant d'IHM

3.1. Introduction

Une fois l'explorateur terminé, j'ai dû travailler à son intégration dans le projet Aloni.

Pour travailler à cette intégration, il m'a fallu comprendre la structure du noyau de l'application telle que l'a spécifiée Pierre-Antoine Bonnin pour pouvoir en utiliser les mécanismes.

Le but de mon intégration n'a pas été de définir l'utilisation concrète de cet explorateur car le reste de l'IHM n'a pas été créé. Il m'a donc fallu créer une couche supplémentaire pour que l'explorateur puisse être utilisé avec les composants du noyau du projet Aloni.

3.2. Structure de données du projet Aloni

Le projet Aloni utilise l'explorateur que j'ai défini dans le but de représenter les deux arborescences présentées dans le paragraphe 1.2.3. Afin de comprendre le fonctionnement de ces deux vues, je vais vous présenter l'architecture définie dans le noyau d'Aloni.

L'élément de base dans l'API d'Aloni est l'interface nommé « IElement ».

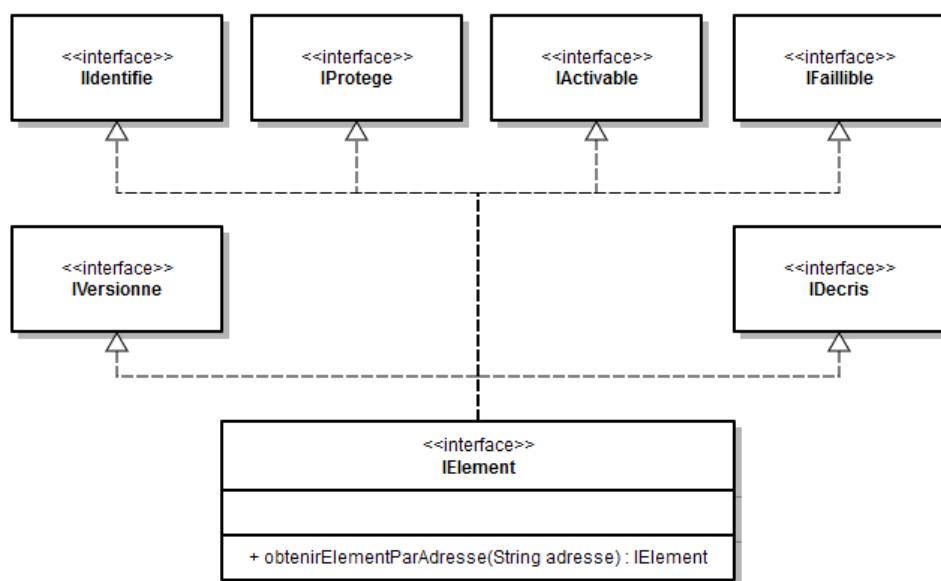


FIGURE 24 - INTERFACE IELEMENT

Cette interface IElement hérite de six interfaces gérant chacune un aspect de son comportement :

- IDecris : interface dédié à l'intitulé et la description de l'élément ;
- IFaillible : interface dédiée à la gestion des erreurs ;
- IVersionne : interface dédiée à la gestion des versions de l'élément ;
- IActivable : interface permettant de définir si l'élément est activable et sa date de péremption ;
- IIdentifie : interface permettant d'associer un identifiant court et un identifiant unique à un élément ;
- IProtege : interface possédant les méthodes nécessaires à la gestion des droits sur cet élément ;

L'interface IElement ajoute à ces six aspects la possibilité de trouver un autre élément par adresse. Cette méthode permet de rechercher un autre élément grâce à une adresse relative à cet élément ou grâce à son identifiant unique.

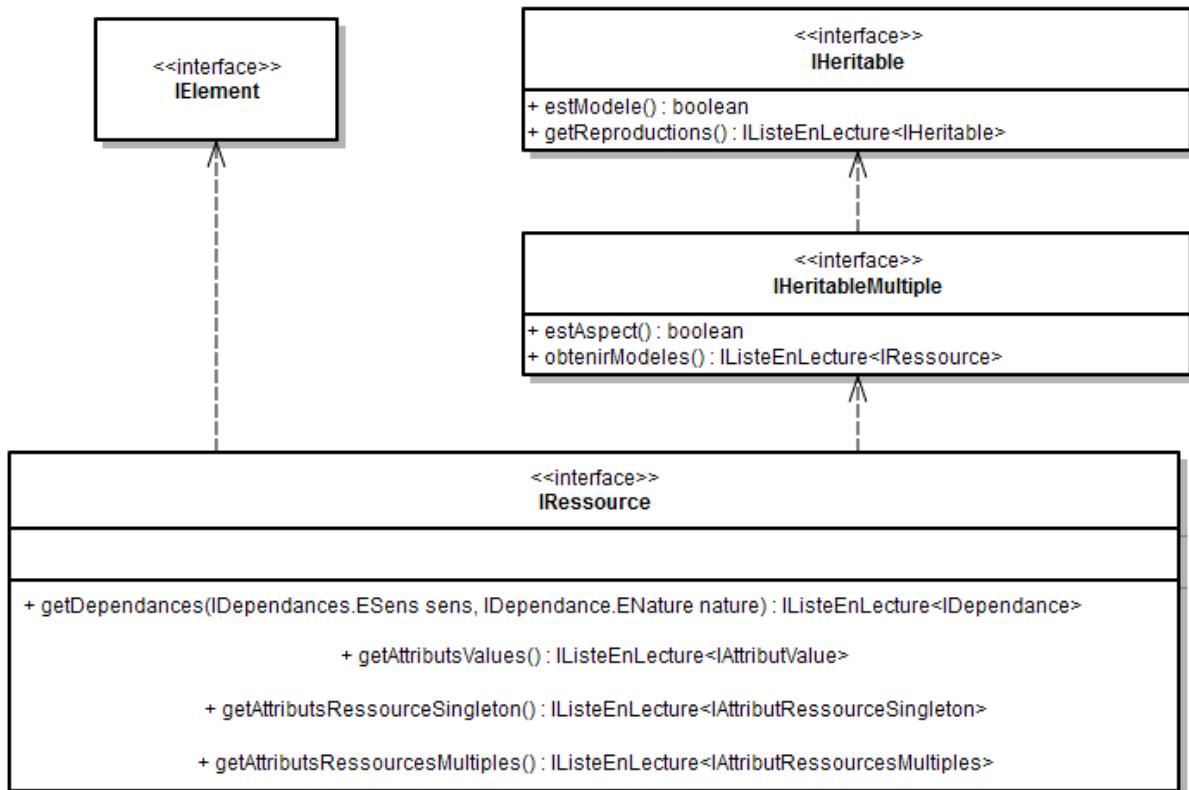


FIGURE 25 - INTERFACE IRESSOURCE

L'interface IResource hérite de l'interface IElement et possède donc l'ensemble des aspects possédés par IElement. Elle hérite également de l'interface « IHitableMultiple ». Cette interface définit quatre méthodes permettant de savoir si la ressource est un modèle, d'obtenir la liste ses modèles, d'obtenir la liste des ressources héritant d'elle si elle-même est un modèle et enfin de savoir si elle est un aspect.

Pour comprendre le concept d'aspect, reprenons l'exemple du « Serveur à Vendre ». Ce modèle de ressource hérite lui-même de deux modèles : le modèle « Serveur » et le modèle « Produit commercial ». Si le modèle « Serveur » peut être directement instancié en une ressource concrète, il ne fait pas de sens d'instancier une ressource « Produit commercial ».

Le modèle « Produit commercial » est donc un aspect, signifiant qu'aucune ressource ne pourrait être directement instanciée depuis sa définition seule.

L'interface IResource permet également d'obtenir les trois listes d'attributs différents que peuvent avoir une ressource : les attributs valués (dont la valeur est de type « simple »), les attributs référençant une seule ressource et les attributs référençant une liste de ressource.

Un attribut, quel que soit sa nature, est toujours lié à une ressource parente que l'on peut obtenir grâce à la méthode « obtenirRessourceParente() ».

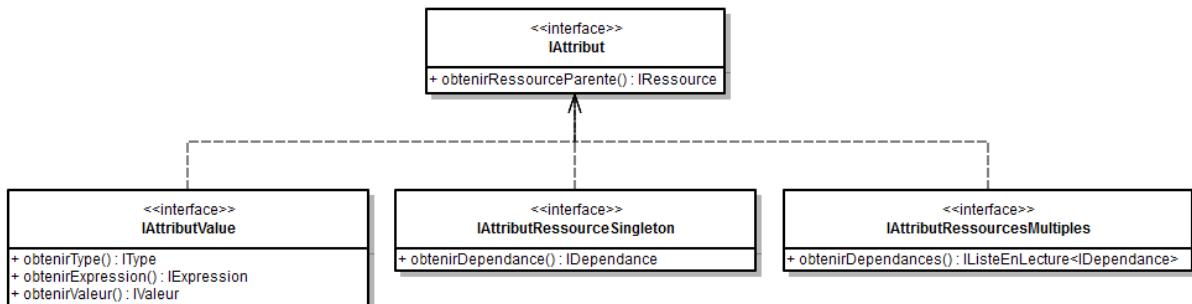


FIGURE 26 - LES INTERFACES IATTRIBUT

Les trois types d'attributs héritent de l'interface « `IAttribut` », elle-même héritant de l'interface « `IElement` » décrite précédemment et de l'interface « `IHeritageSimple` ». `IHeritageSimple`, tout comme `IHeritageMultiple`, propose les méthodes pour savoir si l'attribut est un modèle et pour obtenir la liste des représentations de ce modèle. Cependant, un seul modèle parent peut être défini dans le cadre de l'héritage simple. En effet, un attribut ne peut hériter que de la valeur d'un seul modèle.

Lorsqu'une ressource possède un attribut référençant une ou plusieurs autres ressources, chaque lien entre deux ressources est représenté par un objet héritant de l'interface « `IDependance` ». Cette interface permet d'obtenir le type de dépendance créée (Possession ou référence), la ressource originale du lien, l'attribut par lequel cette dépendance a été créée et la ressource cible du lien.

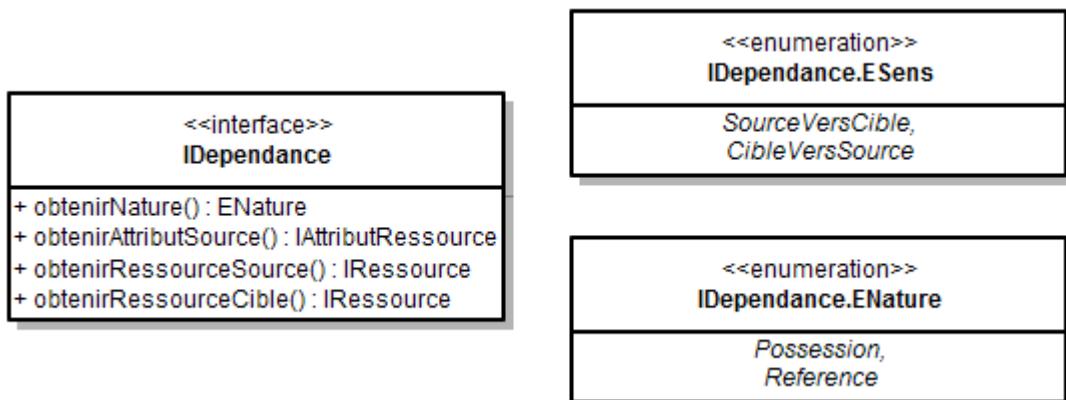


FIGURE 27 - INTERFACE IDEPENDANCE

L'interface « `IRessource` » possède la méthode « `getDependances(IDependance.ESens sens, IDependance.ENature nature)` » qui permet de récupérer l'ensemble des dépendances associées à cette ressource en fonction de leur nature et de leur sens (c'est-à-dire si la ressource sur laquelle la méthode est appelée est source ou cible de la dépendance).

L'interface « `IRessource` » possède également quatre autres méthodes, visible sur la figure de l'interface, qui ne nous intéressent pas pour ce rapport.

3.3. Actions et évènements dans le projet Aloni

Le projet Aloni est destiné à produire une solution utilisée par de nombreuses personnes sur un même déploiement. Il est donc nécessaire de pouvoir définir un système de droits fins pour chaque profil afin de limiter les risques d'erreur ou de failles de sécurité. Cela est mis en pratique grâce à l'utilisation d'un système d'actions dans le noyau du projet. Chaque action de lecture ou de modification de données nécessitera les droits correspondants pour la donnée concernée.

Pour assurer la gestion de ces droits, une logique de séparation des actions a été mise en place, fonctionnant sur la même logique de base que la logique des actions dans l'explorateur, mais de manière plus avancée. Ainsi chaque accès aux données, chaque modification, passe par une action différente, ce qui signifie que les droits du profil utilisateur sont vérifiés pour chaque action.

Le fonctionnement du mécanisme des actions est similaire au fonctionnement présent dans l'explorateur, tout en étant beaucoup plus complet. L'API du noyau d'Aloni prend en compte un certain nombre d'aspects supplémentaires trop complexes et trop peu pertinents pour être expliqués ici (historique, persistance des données, gestion des erreurs, exécution locale et distante...).

La particularité ici est que chaque action peut se voir associer des observateurs. Ainsi, seuls les observateurs concernés sont appelé lorsqu'une action est exécutée.

Je me suis donc appuyé sur ces observateurs et ses actions pour définir le comportement de la surcharge.

3.4. Spécification de la surcharge

Le but de cette surcharge est de créer une interface ergonomique et facilement compréhensible pour manipuler un nombre potentiellement très important de données.

Pour cela, il m'a fallu créer deux types de vues différentes : la vue sur l'arborescence d'héritage des modèles de ressource et la vue sur l'arborescence d'appartenance des instances de ressources.

Dans chacune des deux vues, nous retrouvons des conventions communes afin de créer une cohérence.

Une ressource sera représentée par un Item que j'ai défini pour ressembler à la capture suivante.



FIGURE 28 - ITEM REPRESENTANT UNE RESSOURCE

Une police grasse, l'absence de préfixe, et l'icône de boîte servent à signifier qu'il s'agit de l'intitulé d'une ressource. On peut aussi remarquer que les ressources possèdent une liste de détails concernant leur version.

Tous les attributs sont symbolisés par une arobase et un préfixe de couleur grise indiquant le nom de l'attribut.



FIGURE 29 - ITEM REPRESENTANT UN ATTRIBUT VALUE

Les attributs valués possèdent une valeur visible en texte noir, tandis que les attributs contenant une ou plusieurs sous-ressources ne contiennent pas de valeur mais possède en tant que sous éléments les ressources qu'ils réfèrentent.



FIGURE 30 - ITEM REPRESENTANT UN ATTRIBUT DE TYPE RESSOURCE

Les ressources modèles, symbolisés par la sur-icône « M », possèdent trois types de sous éléments : leurs attributs modèles, les modèles étendant leur définition et les liens vers les instances de ressource héritant de ces modèles.



FIGURE 31 - EXEMPLE DE HIERARCHIE DE MODELE

Dans la figure ci-dessus, le modèle « Application Web », héritant du modèle « Déploiement », possède quatre attributs modèles. Le fait que l'attribut modèle « Routeur » ai comme sous-élément le modèle de ressource « Routeur » signifie que seul des instances héritant du modèle « Routeur » pourront être acceptés comme valeur de l'attribut « Routeur » dans une instance de « Application Web ».

On voit également un lien vers la ressource concrète appelée « Projet Toto – Déploiement Rennes », ce qui signifie que cette ressource hérite de « Application Web ».

Voici les deux vues correspondant à un jeu de données utilisé pour une démonstration :

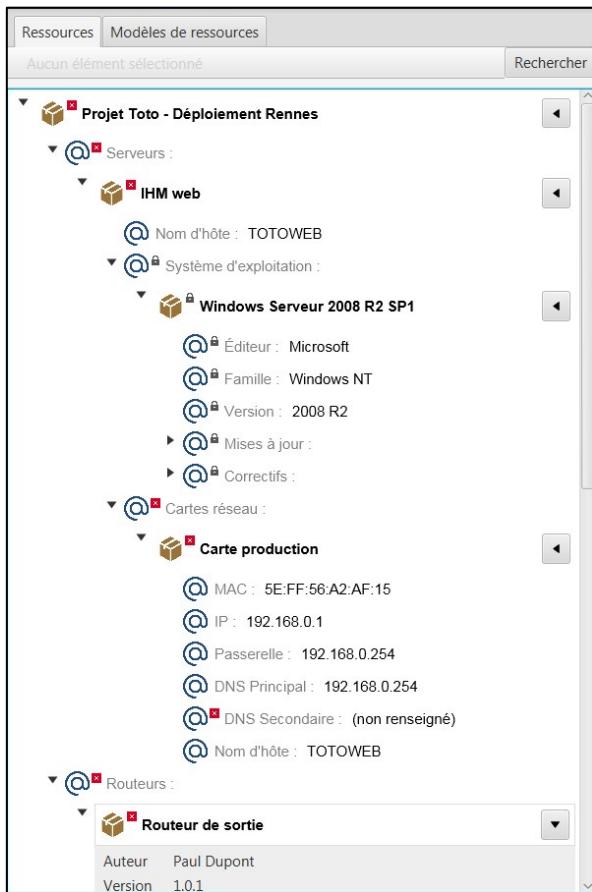


FIGURE 32 - VUE DES INSTANCES DE RESSOURCES

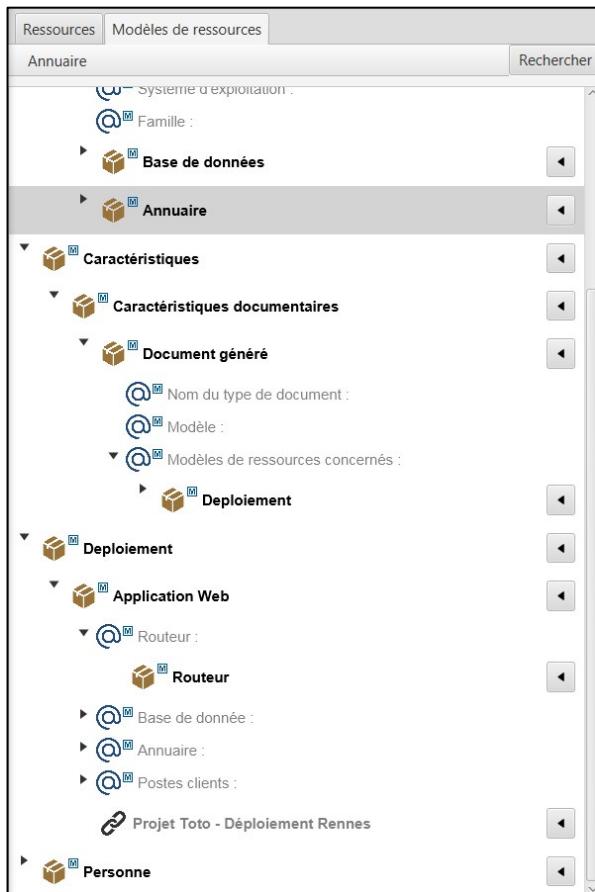


FIGURE 33 - VUE DES MODELES DE RESSOURCES

L'intérêt d'avoir ces deux vues est également de pouvoir naviguer de l'une à l'autre. Pour cela, j'ai ajouté les actions correspondantes sur les éléments.



FIGURE 34 - ACTION "INSPECTER LE MODELE"



FIGURE 35 - ACTION "INSPECTER CETTE RESSOURCE"

Ces actions changent d'onglet sélectionné pour mettre en avant l'autre vue et sélectionne l'élément en question dans l'autre vue.

3.5. Conclusion de la deuxième partie

Je n'ai pas terminé la spécification de l'interface entre mon composant d'explorateur et le noyau Aloni, et je n'aurais probablement pas le temps de la terminer avant la fin de mon stage. J'ai dû néanmoins présenter une version de démonstration le mardi 31 mai aux clients de la DGA-MI. Ils se sont dit satisfaits de ma présentation, et m'ont donné des pistes afin d'améliorer l'ergonomie de mon explorateur. Les considérations ergonomiques ont été prises en compte depuis. Elles concernaient notamment la recherche que j'ai présentée précédemment et le fait de pouvoir organiser les actions associées à un nœud en groupes ou par dossiers.

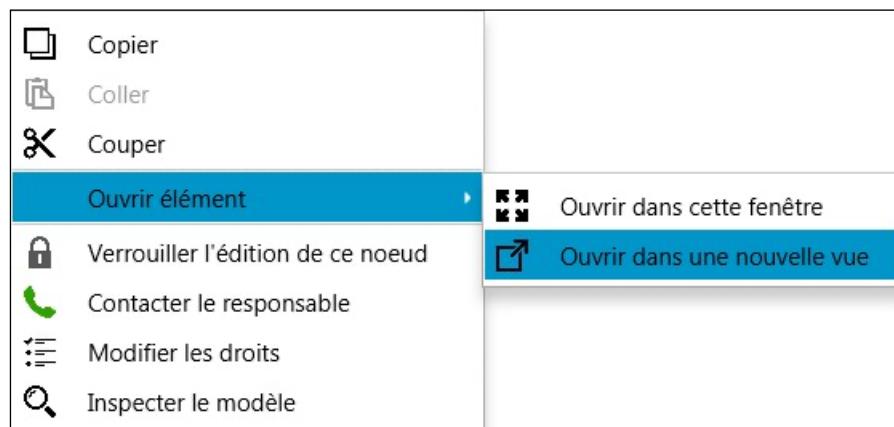


FIGURE 36 - EXEMPLES DE SEPARATEURS ET DE DOSSIERS DANS LES MENUS D'ACTIONS

La fin de mon stage constituera donc en un « nettoyage » de mon code source et un passage de connaissance à la personne qui sera en charge de poursuivre mon travail.

4. Conclusion

J'ai eu la chance de faire mon stage de fin d'études dans une entreprise ayant une politique d'accueil des stagiaires permettant une réelle responsabilisation. Les missions confiées aux stagiaires sont similaires aux missions que remplissent les salariés, tout en gardant la mesure de la différence d'expérience et de compétences.

Cette responsabilisation m'a beaucoup plu, et j'ai réellement apprécié travailler au sein d'un projet concret, en contact avec le client. Je pense avoir réussi à développer un composant répondant aux attentes de mon tuteur de stage et je tire une certaine fierté à l'idée que mon travail sera utilisé pour d'autres projets après mon départ.

Cela dit, ce qui a fait de ce stage une réussite à mes yeux n'est pas seulement le travail accompli, mais tous les apprentissages que j'ai pu en retirer. J'ai notamment beaucoup appris sur la théorie de la programmation objet pendant ce stage et je pense que ce sera un bénéfice considérable à l'avenir.

Je conclurais en ajoutant que j'ai beaucoup apprécié travailler à Capgemini Rennes grâce à l'ambiance chaleureuse que l'on peut y retrouver. J'ai pleinement conscience de ma chance d'avoir effectué un stage de si bonne qualité dans un environnement si agréable, et je remercie une dernière fois toutes les personnes qui ont rendu cela possible.

5. Glossaire

API : Acronyme pour « Applications Programming Interface ». Une API est une interface de programmation qui permet de « brancher » deux applications ou deux parties d'un code sources entre elles.

Capacités : A l'inverse des fonctionnalités qui désignent les services rendus à l'utilisateur par une application, les capacités désignent la manière dont ces services sont rendus. Elles désignent le plus souvent l'optimisation de l'application ou sa capacité à fonctionner dans tous type de contexte.

Constructeur : Ici, méthode définissant les opérations à effectuer à la création d'un objet.

COTS : Acronyme pour « Commercial Off-The-Shelf ». Désigne les logiciels que l'on peut trouver dans le commerce, en opposition aux logiciels réalisés par une SSII pour une autre entreprise.

DGA-MI : Acronyme pour « Direction Générale de l'Armement – Maitrise de l'Information ». La DGA-MI est un centre d'expertise technique de la recherche pour la défense française, qui a pour mission des études, expertises et essais dans les domaines de la guerre électronique des systèmes d'armes, des systèmes d'information, des télécommunications, de la sécurité de l'information et des composants électroniques.

Déploiement : Installation d'un système, d'une application, d'un service information ou toute infrastructure.

IHM : Acronyme pour « Interface Homme-Machine ». Désigne l'interface graphique d'une application ou d'une page web.

JavaFx : Bibliothèque graphique pour Java.

Noyau : Ici, désigne le code source d'une application ou d'une partie de l'application réalisant la majeure partie des opérations, et n'apparaissant pas dans l'API.

Vue : Ici, désigne une partie de l'IHM permettant de visualiser le jeu de données.

Solution technique : Désigne une application, un service web, ou tout autre solution informatique répondant au besoin du client.

Surcouche : Désigne ici la partie du code rajoutée à une librairie externe pour en faciliter l'usage que l'on souhaite en faire.

SSII : Acronyme pour « Société de services et d'ingénierie en informatique ». Désigne une société experte dans le domaine des nouvelles technologies et de l'informatique.

6. Table des figures

Figure 1 - Organisation dans un projet de développement informatique	4
Figure 2 - Organisation des phases d'un déploiement	8
Figure 3 - Composants communs aux différentes phases d'un déploiement.....	9
Figure 4 - Exemple d'héritage	11
Figure 5 - Démonstration de la multiplicité des vues	16
Figure 6 - Exemple de multiplicité des parents.....	16
Figure 7 - Structure d'un élément de l'arborescence	18
Figure 8 - Exemple d'utilisation des sur-icônes.....	18
Figure 9 - Exemple de sur-icone non contaminante	19
Figure 10 - Exemple de modification rapide d'une valeur	19
Figure 11 - Actions dans la barre de menu	20
Figure 12 - Actions dans le menu contextuel.....	20
Figure 13 - Exemple de noeud "Lien"	20
Figure 14 - Exemple d'utilisation de la pagination.....	21
Figure 15 - Affichage d'arborescence de JavaFx	22
Figure 16 - Organisation du noyau de l'explorateur	24
Figure 17 - Interface IExplorateur	25
Figure 18 - Interface IObservateur.....	26
Figure 19 - Interfaces IAction et IContexte	27
Figure 20 - Interfaces utilisées pour définir une recherche.....	28
Figure 21 - Présentation du panneau de recherche	29
Figure 22 - Relations entre les quatre classes principales du noyau de l'explorateur	31
Figure 23 - Séquence d'initialisation des aspects	32
Figure 24 - Interface IElement	35
Figure 25 - Interface IRessource	36
Figure 26 - Les interface IAttribut	37
Figure 27 - Interface IDependance	37
Figure 28 - Item représentant une ressource	38
Figure 29 - Item représentant un attribut value	39
Figure 30 - Item représentant un attribut de type ressource.....	39
Figure 31 - Exemple de hiérarchie de modèle	39
Figure 32 - Vue des instances de ressources	40
Figure 33 - Vue des modèles de ressources.....	40
Figure 34 - Action "Inspecter le modèle" Figure 35 - Action "Inspecter cette ressource"	40
Figure 36 - Exemples de séparateurs et de dossiers dans les menus d'actions.....	41

Note : cette page doit se situer en QUATRIEME DE COUVERTURE au dos du rapport

Résumé

La formation du DUT Informatique se termine par une période de stage en entreprise de dix semaines. J'ai eu la chance d'effectuer mon stage au sein de la branche Rennaise de Capgemini, un leader mondial dans la catégorie des SSII.

J'ai pu contribuer au projet d'étude Aloni, pour le compte de la DGA-MI. Ce projet a pour but de créer un outil d'aide à la planification des déploiements. J'ai réalisé un composant d'IHM de cette solution, permettant d'afficher et de manipuler des arborescences.

Ce stage m'a initié à la librairie graphique JavaFx, ainsi qu'à de nombreux concepts liés à la programmation orientée objet. J'ai également pu participer à deux réunions avec les interlocuteurs en charge du projet Aloni à la DGA-MI.

MOTS CLES : déploiement, IHM, JavaFX

Summary

The formation to obtain the technology degree in computer science ends with a ten weeks internship in a company. I was lucky enough to do my internship in the branch of Gapgemini located in Rennes. Capgemini is one of the international leaders in the IT service field.

I have been able to contribute to the study project Aloni, on the behalf of the DGA-MI. The goal of this project is to create a tool helping at the planning of deployments. I worked on the IHM component of this solution, allowing to display and manipulate tree views of data.

This internship made me discover the graphical library JavaFx as well as a lot of concepts concerning the object oriented programming. I also have been part of two meetings with the representative in charge of the Aloni project at the DGA-MI.

KEY WORDS : deployments, IHM, Java FX