MODELISATION ORIENTE-OBJET D'ASPECTS OPERATIONNELS DE BASE DE DONNEES SIDERURGIQUE

Yves Wautelet, Laurent Louvigny, Manuel Kolp
IAG – ISYS (Unité de Systèmes d'Information), Université Catholique de Louvain,
1 Place des Doyens, 1348 Louvain-la-Neuve, Belgique
Contact: kolp@isys.ucl.ac.be

Résumé: Ce document constitue une étude orienté objet de l'entreprise sidérurgique Carsid au cours de laquelle une modélisation de la cokerie, de la traction et de la bascule a été effectuée. Pour ce faire, le langage de modélisation UML a principalement été utilisé. Les différentes phases du développement en UML peuvent être représentées au moyen d'une série de diagrammes permettant de comprendre de manière visuelle les concepts définis. Tous les modèles s'enchaînent en passant de l'analyse à la conception, gagnant en complexité, dans un langage commun et unique, s'affinant au fur et à mesure pour arriver à l'élaboration finale du modèle. Les diagrammes permettront de comprendre sous différents angles la globalité du cas étudié en présentant une vue statique et dynamique de celui-ci. Chaque diagramme exprimera une partie de la structure totale, tout en étant un aspect particulier du modèle.

1 INTRODUCTION

Les systèmes d'information prennent une place de plus en plus importante en entreprise, et les répercussions de cette évolution sont nombreuses. Maîtriser habilement les nouvelles technologies peut apporter l'avantage concurrentiel qui permettra à l'entreprise de prospérer.
Cependant, de nombreux projets en technologie de l'information ne voient jamais le jour, et même ceux qui arrivent à terme sont la plupart du temps sous-utilisés à cause d'une mauvaise prise en compte des besoins de leurs utilisateurs.
Dès lors, la modélisation orienté-objet se révèle un outil intéressant dans l'élaboration de tels projets. Une bonne modélisation permet en effet de s'assurer que les concepteurs du système informatique et les utilisateurs s'entendent sur les spécifications de ce système. D'autre part, le modèle est indépendant du langage informatique utilisé par la suite. Enfin, il est évolutif, et peut être utilisé tout au long d'un projet, et adapté en fonction des nouveaux besoins rencontrés.

Carsid (Carolo-Sidérurgie) est une joint venture récente issue du mouvement de concentration. Elle a été crée par les sociétés Duferco et Usinor Belgium S.A. (composition du capital de 60/40). Duferco est une entreprise italienne qui est notamment active en Belgique (la Louvière et Clabecq). Usinor fait partie du géant mondial de l'acier Arcelor, en partenariat avec le luxembourgeois Arbed et l'espagnol Acelaria. Duferco possède une part majoritaire dans la société Carsid.

Pour une entreprise sidérurgique comme Carsid, le développement de systèmes informatiques toujours plus performants est certainement une des solutions pour mieux résister à la concurrence. De fait, des producteurs d'acier issus de pays où le coût de la main d'œuvre est plus bas font peser une pression constante sur notre industrie sidérurgique, qui a pour seules parades une consolidation généralisée du secteur, une spécialisation dans les aciers plus techniques et une augmentation de l'efficacité de la production.

Le projet sur la cokerie de Carsid, qui a vu le jour grâce à la collaboration entre l'entreprise Carsid et l'Unité de Systèmes d'Information (ISYS) de l'Université catholique de Louvain, est une tentative qui cadre pleinement avec cette augmentation d'efficacité. Son objectif est de moderniser et de standardiser les systèmes d'information actuels pour une meilleure exploitation des bases de données existantes.

Le travail qui suit présente une modélisation orienté objet de plusieurs processus opérationnels de

l'entreprise Carsid. Il s'intéresse plus particulièrement aux besoins des utilisateurs du système informatiques de la cokerie de Marchienne et au fonctionnement de la traction et de la bascule. Ces derniers représentent les dispositifs de gestion des entrées et des sorties de matières de l'entreprise pour les transports par rail et par camion.

Dans ce document, le langage de modélisation UML est tout d'abord présenté. Ensuite, une modélisation de type « gestion des exigences » pour la cokerie et une modélisation de type « modélisation métier » pour la traction et la bascule y sont abordés. Pour réaliser ces différents travaux, un diagramme des cas d'utilisation et des diagrammes d'activité ont été définis pour chaque entité modélisée. Enfin, dans l'activité d'analyse et de conception, le diagramme de classe présente l'architecture du futur système et des diagrammes de séquence mettent en perspective l'utilisation des méthodes qui y sont définies. Une spécification de ces méthodes se trouve également dans l'annexe de ce document.

2 ELEMENTS DE MODELISATION

La modélisation est une des tâches les plus importantes dans le processus de développement d'un système. La phase consacrée à l'analyse peut être considérée comme plus stratégique que celles dévolues à la conception et l'implémentation proprement dites. Il faut en effet fondamentalement représenter, comprendre et identifier les exigences du système afin de concevoir puis d'implémenter une application stable et performante.
Avec l'augmentation de la complexité des systèmes à élaborer, le choix d'une méthode de développement appropriée se révèle primordial pour le succès des travaux. Pour cela, il existe plusieurs orientations disponibles, c'est-à-dire des approches différentes pour comprendre, représenter, analyser et concevoir un système. Le problème sera décomposé en plusieurs modèles différents liés entre eux.

Modélisation. L'élaboration d'un système informatique passe toujours par une phase de modélisation préalable afin d'en réduire la complexité et pouvoir simuler la réalité. Un modèle est une abstraction de la réalité et la modélisation un processus qui consiste à identifier les caractéristiques pertinentes d'une entité, en vue d'une utilisation précise.
L'utilisation d'un modèle présente plusieurs avantages :

- un modèle peut être utilisé lors de simulations, pour mieux comprendre la chose qu'il représente ;
- un modèle peut évoluer à mesure que l'on comprend mieux la tâche ou le problème ;
- on peut décider quels détails inclure ou ignorer dans le modèle.

Orienté objet. Dans l'approche orienté-objet, les systèmes sont uniquement constitués d'entités appelées *objets*. Un objet est « *une abstraction d'une chose du problème reflétant la capacité du système à garder de l'information sur cette chose et/ou à interagir avec elle* ».
Ces objets sont définis par des *types* qui définissent de façon syntaxique et sémantique les propriétés que présenteront les objets du type ; ces propriétés permettent de délimiter, de qualifier les objets et de savoir comment on peut communiquer avec eux. L'implémentation de ces propriétés est ensuite fournie par une *classe*, structure de données qui lie à une propriété une implémentation possible. L'implémentation des propriétés par une classe peut être représentée soit sous forme d'emplacement mémoire (champs de donnée) appelé *attribut*, soit sous forme de calcul (opération) appelé *méthode*.
En orienté-objet, si un type définit donc l'interface de l'objet, la classe lui fournit une implémentation. A ce titre, la classe est le moule par lequel est construit un objet. On dit alors d'un objet qu'il est une *instance* de telle classe. Les objets interagissent via des envois de message.
L'approche orienté-objet apporte toute une série d'atouts à une modélisation adéquate. Les solutions obtenues sont en effet indépendantes des modifications physiques de l'architecture et donc plus facilement adaptables aux évolutions du système. Il sera possible d'étendre les possibilités de traitement des logiciels en ajoutant aux schémas existants de nouveaux objets mettant à jour la fonctionnalité globale des modèles.
Un langage de modélisation unifié est apparu il y a une dizaine d'années : UML.

UML, langage de modélisation. UML (« Unified Modeling Language ») est un langage de conception se basant sur la création de modèles successifs de plus en plus détaillés pour mettre en place une solution au problème étudié. UML a succédé à une série de méthodes d'analyse et de conception

orienté-objet au début des années 90. Il est la fusion et la synthèse des trois méthodologies dominantes de Booch (OOT), de Rumbaugh (OMT) et de Jacobson (OOSE), bien que sa portée soit beaucoup plus vaste. En 1997, UML a subi un processus de normalisation de la part de l'OMG (« Object Management Group »), le consortium chargé de définir les standards de l'industrie, et est aujourd'hui devenu incontournable.
UML est le fruit d'un travail d'experts reconnus ; il est issu du terrain, ce qui est un atout indéniable. De plus il est riche car il couvre toutes les phases d'un cycle de développement et il est ouvert car indépendant du domaine d'application et des langages d'implémentation. Etant le résultat d'un large consensus parmi les méthodologistes et les industriels, il s'impose aujourd'hui comme le standard de l'industrie.
Enfin, il existe de nombreux outils qui supportent UML : Rational Rose d'IBM, Together de Borland, Visio Pro de Microsoft, etc.
Les différentes phases du développement UML peuvent être représentées au moyen d'une série de diagrammes permettant de comprendre de manière visuelle les concepts définis. Tous les modèles s'enchaînent en passant de l'analyse à la conception, gagnant en complexité, s'affinant au fur et à mesure pour arriver à l'élaboration finale du modèle. Les diagrammes permettent de comprendre sous différents angles la globalité du cas étudié en présentant une vue statique et dynamique de celui-ci. Chaque diagramme exprime une partie de la structure totale, tout en étant un aspect particulier du système.
Ces diagrammes sont répartis en trois catégories. Chacune de ces catégories apporte une approche différente du modèle. Les diagrammes *statiques* permettent de décrire les aspects structurels d'un système, c'est-à-dire les éléments qui le composent. Les diagrammes *dynamiques* décrivent plutôt les aspects comportementaux du système, c'est-à-dire la manière dont un système réagit en fonction de certains événements ou actions. Enfin, les diagrammes d'*implémentation* décrivent les éléments nécessaires au déploiement de ce système lors de l'implémentation.

3 METHODOLOGIE : LES ACTIVITES DE MODELISATION

Modélisation métier.
La modélisation métier a pour but :

- de décrire la structure et la dynamique de l'organisation dans laquelle le système est déployé ;
- de comprendre les problèmes courants dans l'organisation et d'identifier les améliorations potentielles ;
- de garantir que les clients, les utilisateurs finaux et les développeurs partagent une vision commune de l'organisation ;
- de réaliser une base d'informations qui contiendra le cahier des charges du produit et la planification des tâches de l'organisation.

Pour atteindre ces buts, la modélisation métier décrit comment développer une vision de la nouvelle organisation et, basé sur cette vision, définit les processus, rôles et responsabilités de cette organisation dans le modèle de l'entreprise.

Gestion des exigences.
La gestion des exigences a pour but :

- d'établir et de maintenir les accords avec les clients et autres *stakeholders* sur ce que le système doit faire ;
- de procurer aux développeurs une meilleure compréhension des besoins du système ;
- de définir les limites du système ;
- de procurer une base pour planifier le contenu technique des itérations ;
- de procurer une base pour estimer le coût et le temps pour développer le système ;
- de définir et de construire une maquette de l'interface utilisateur basée sur les exigences et les buts de celui-ci.

Pour atteindre ces buts, la gestion des exigences décrit comment définir une vision du système et traduire la vision en un modèle des cas d'utilisation accompagné de spécifications externes constituant le cahier des charges logicielles. De plus, la gestion des exigences décrit comment utiliser les attributs des exigences pour aider à gérer la portée et le changement d'exigences du système.

Analyse et conception
L'analyse et la conception ont pour but :

- de comprendre le cahier des charges et d 'écrire les spécifications internes qui décrivent comment implémenter le système (exigences transformées dans la meilleure stratégie d'implémentation possible). L'analyse permet d'obtenir une vue interne idéale du système ;
- la conception a pour but de définir une architecture robuste du système (c'est-à-dire facile à comprendre, construire et faire évoluer) qui recouvre entièrement les besoins de celui-ci ;
- l'analyse se concentre sur le "quoi faire", la conception se concentre sur le "comment le faire".

Implémentation.
L'implémentation a pour but :

- de définir l'organisation du code en terme de sous-systèmes d'implémentation organisés en couches ;
- d'implémenter classes et objets en terme de composants ;
- de tester les composants développés comme des unités ;
- d'intégrer dans un système exécutable les résultats produits par des programmeurs individuels ou des équipes.

Test.
La phase de test a pour objectif d'évaluer le niveau de qualité atteint par le produit et d'en tirer les conclusions. Ceci ne comprend pas uniquement le produit fini, mais commence tôt dans le projet avec la validation de l'architecture et continue à travers la validation du produit fini au consommateur.
Les tests comprennent :

- la vérification des interactions des composants ;
- la vérification de la bonne intégration des composants ;
- la vérification que toutes les exigences ont été implémentées correctement ;
- l'identification et la vérification que toutes les défectuosités découvertes sont corrigées avant le déploiement du logiciel.

Déploiement.
Le but des activités de déploiement est de livrer le produit aux utilisateurs finaux.

4 GESTION DES EXIGENCES DE LA COKERIE

Une *modélisation métier* de la cokerie a été réalisée par Aymerick Donnay et les membres du département ISYS lors de la première phase du projet Carsid. L'objectif de cette modélisation métier était de décrire la structure et la dynamique de l'organisation dans laquelle le système est déployé, et ne peut donc être la seule base des activités d'implémentation. Une analyse plus proche des systèmes existants et des besoins des futurs utilisateurs devait donc compléter cette première modélisation. Cette activité, nommée *gestion des exigences* a été réalisée par Laurent Louvigny et Yves Wautelet.

Lors de la gestion des exigences un nouveau diagramme de cas d'utilisation à été définit. Chaque cas d'utilisation est décrit et accompagné d'un diagramme d'activités qui s'y rapporte.
Dans la première section, les concepts propres à la cokerie sont définis. Ensuite, dans la seconde section, la modélisation proprement dite est présentée.

4.1 Les concepts

Afin d'illustrer certains concepts, tels l'enfourneuse, le four, la défourneuse et le coke car, nous reprendrons des diagrammes d'états-transitions générés lors de la première phase du projet Carsid.

La pâte à coke. Les charbons, qui proviennent principalement des Etats-Unis, d'Australie, d'Afrique du Sud et de Pologne, sont de qualité et de granulométrie variables en fonction de leur origine. Ils doivent donc passer dans une installation où ils sont homogénéisés et broyés afin de préparer un mélange qui soit le plus constant possible, appelé *pâte à coke.*

L'enfourneuse. Avant la cuisson, il faut commencer par remplir les fours de pâte à coke. On utilise pour cela une machine appelée *enfourneuse* qui se déplace sur des rails placés au-dessus des fours pour les remplir par des orifices situés sur leur partie supérieure. La charge d'un four est d'environ 16 tonnes. Il existe à Carsid deux types d'enfourneuses. L'enfourneuse E1, plus ancienne, et l'enfourneuse E3 qui grâce à des automatismes fournit des données de production : heure, numéro du four, poids et humidité de la pâte. La Figure 1

représente les différents états par lesquels passe l'enfourneuse durant ses activités.

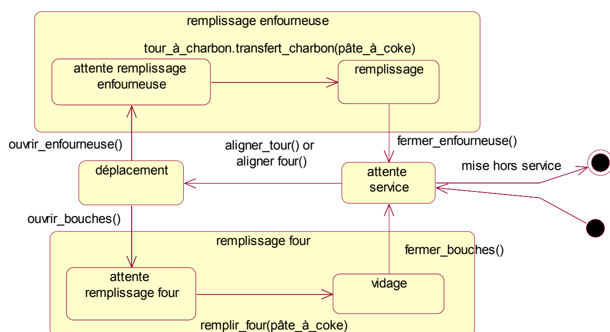

Figure 1 : diagramme d'états-transitions de l'enfourneuse.

Le four, le piedroit et le carneau. La pâte à coke est distillée à l'abri de l'air dans des cellules de *fours* construites en briques réfractaires et entourées de *piédroits*. Les piédroits sont des murs creux à l'intérieur desquels brûle du gaz qui porte la température des parois réfractaires aux environs de 1.300°C, ces parois communiquant alors leur chaleur au charbon. La combustion du gaz est réalisée dans des brûleurs, appelés *carneaux*, répartis sur toute la longueur du piédroit. Le coke, d'un rouge incandescent à sa sortie, est alors appelé *saumon de coke*.
L'installation à Carsid comprend un total de 122 fours (Figure 2).

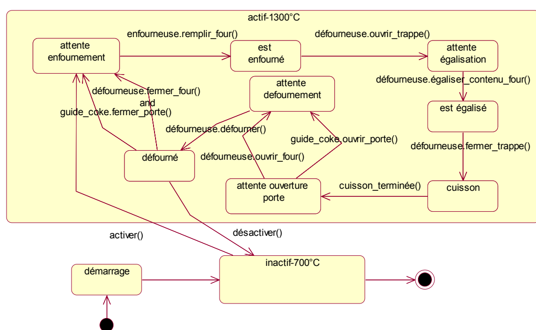

Figure 2 : diagramme d'états-transitions du four

La défourneuse. Après la cuisson, il faut extraire le saumon de coke du four. La machine utilisée est appelée *défourneuse*, et son rôle est d'ouvrir la porte latérale du four et de pousser le coke. Il y a trois défourneuses chez Carsid, deux en service (la D1 et la D3) et une en réserve (la D2). Les états d'une défourneuse sont détaillés à la Figure 3.

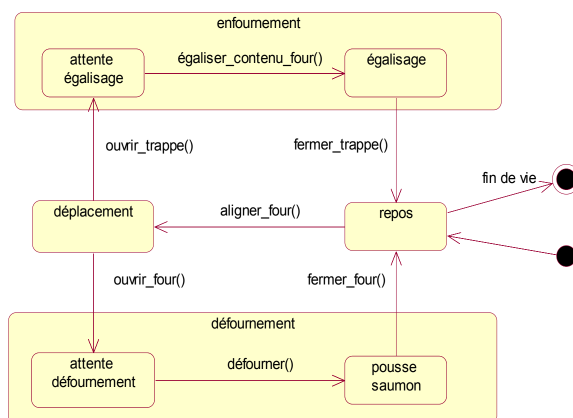

Figure 3 : diagramme d'états-transitions de la défourneuse

Le guide coke et le coke car. Le *guide-coke* est une sorte de couloir mobile assez court par lequel passe le saumon poussé par le bras mobile de l'enfourneuse. Le saumon de coke se déverse alors dans un wagon, le *coke-car*. Ce dernier conduit ensuite le coke défourné sous une tour d'extinction où il est refroidi par aspersion brutale d'eau (Figure 4).

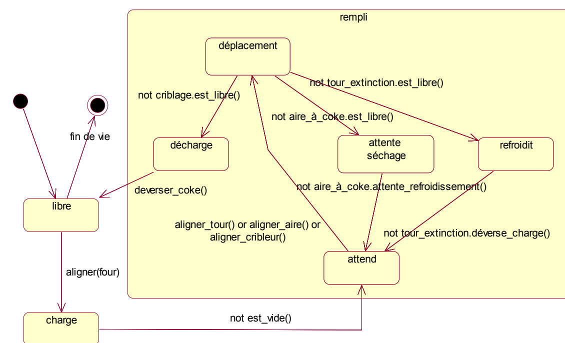

Figure 4 : diagramme d'états-transitions du coke-car

Le cahier électronique. Le *cahier électronique* est une page de l'Intranet de Carsid sur laquelle l'équipe contrôle de la cokerie peut enregistrer toute une série d'informations (température, pression, vitesse, etc.).

Le planning défournement. Le *planning défournement* est très important pour le bon

fonctionnement de la cokerie. En effet, les fours doivent toujours rester au-dessus d'environ 700°C, sans quoi les briques réfractaires cassent. Il faut donc assurer une tournante régulière et précise des cuissons. Pour éviter les chocs thermiques, les fours doivent aussi être enfournés par pas de cinq.
Le planning défournement est la planification de tous les enfournements et défournements relatifs à une *pause* (comme l'activité de la cokerie est incessante, les équipes y travaillent par pauses, c'est-à-dire par roulement). C'est le contremaître qui est chargé de créer ce planning, en utilisant l'Intranet de Carsid.
L'allure de marche, imposée par le temps de cuisson, nécessite 168 enfournements journaliers.

L'intranet. L'*Intranet* de Carsid est un outil qui permet d'encoder toute une série de données, d'utiliser des applications et de rechercher des informations, par exemple sous forme de graphiques. Différentes utilisations de l'Intranet seront rencontrées lors de la modélisation de la cokerie.

4.2 La modélisation

La gestion des exigences prend en compte les besoins exprimés par les différents *stakeholders*. Un diagramme des cas d'utilisation basé sur les besoins des utilisateurs du système informatique a ainsi été développé.
Dans cette démarche, la première étape est de définir les acteurs physiques devant utiliser le système informatique et les acteurs machine permettant de relever des informations qui, une fois introduites dans le système informatique, peuvent être analysées.

Les utilisateurs interagissent avec le système informatique via le biais de l'intranet. Chaque corps de métier utilisant le système peut donc être vu comme un utilisateur de celui-ci. Ces différents corps de métier, généralisés par l'entité *utilisateur*, sont :

- l'équipe réglage ;
- le management ;
- le service informatique ;
- les ingénieurs de production ;
- le contremaître ;
- l'équipe contrôle.

Les outillages impliqués dans des interactions avec le système informatique sont :

- les enfourneuse E1 et E3 ;
- la défourneuse ;
- le guide coke ;
- les fours.

Description des cas d'utilisation.

Utiliser intranet
Objectif : utiliser le système informatique via le biais de l'intranet.
Description : cas d'utilisation générique représentant l'utilisation du système informatique via l'intranet par toute une série d'acteurs particuliers à des fins différentes représentées dans les cas d'utilisation spécifiques décrits ci-après.
Intervenants : équipe réglage, management, service informatique, ingénieurs de production, contremaître, équipe contrôle.
Cas d'utilisation associés : rechercher information, encoder température carneaux, créer planning défournement, encoder cahier électronique.

Rechercher information
Objectif : rechercher des informations utiles à la prise de décision.
Description : le management, le service informatique et les ingénieurs de production utilisent l'intranet pour rechercher les informations dont ils ont besoin. Les informations transmises constituent un formatage (graphique, tableau) des données fournies par la base de données.
Intervenants : management, service informatique, ingénieurs de production.
Cas d'utilisation associé : utiliser intranet.

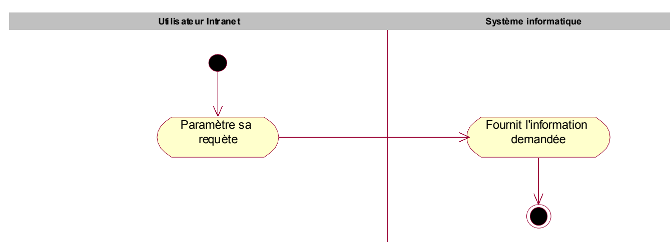

Figure 5 : Diagramme d'activités pour *Rechercher Informations*

Encoder température carneaux
Objectif : encoder dans le système informatique la température relevée dans les carneaux.

Description : l'équipe réglage mesure la température dans les carneaux durant la cuisson et les encode ensuite dans la base de données via l'intranet.
Intervenant : équipe réglage.
Cas d'utilisation associés : utiliser intranet, cuisson.

Créer planning défournement
Objectif : créer le planning des enfournements et défournements de la pause suivante.
Description : un *planning défournement* est la planification de tous les enfournements et de tous les défournements pour la pause suivante. A chaque fin de pause, le contremaître utilise l'intranet pour en créer un nouveau. Pour ce faire, il fait exécuter au système informatique un algorithme de *branch and bound* (le modèle Patricio Villa) qui lui fournit une proposition de planification des enfournements et défournements pour la pause suivante si une telle solution a été trouvée. Si cet algorithme n'a pu trouver de solution ou que le contremaître le souhaite, il peut exécuter le modèle light qui est un modèle dégradé qui fournit une planification des enfournements et des défournements sous-optimale (solution *par défaut*). Dans les deux cas, le contremaître peut faire des modifications sur le planning qui lui est proposé et doit valider le planning qu'il l'ait modifié ou non.

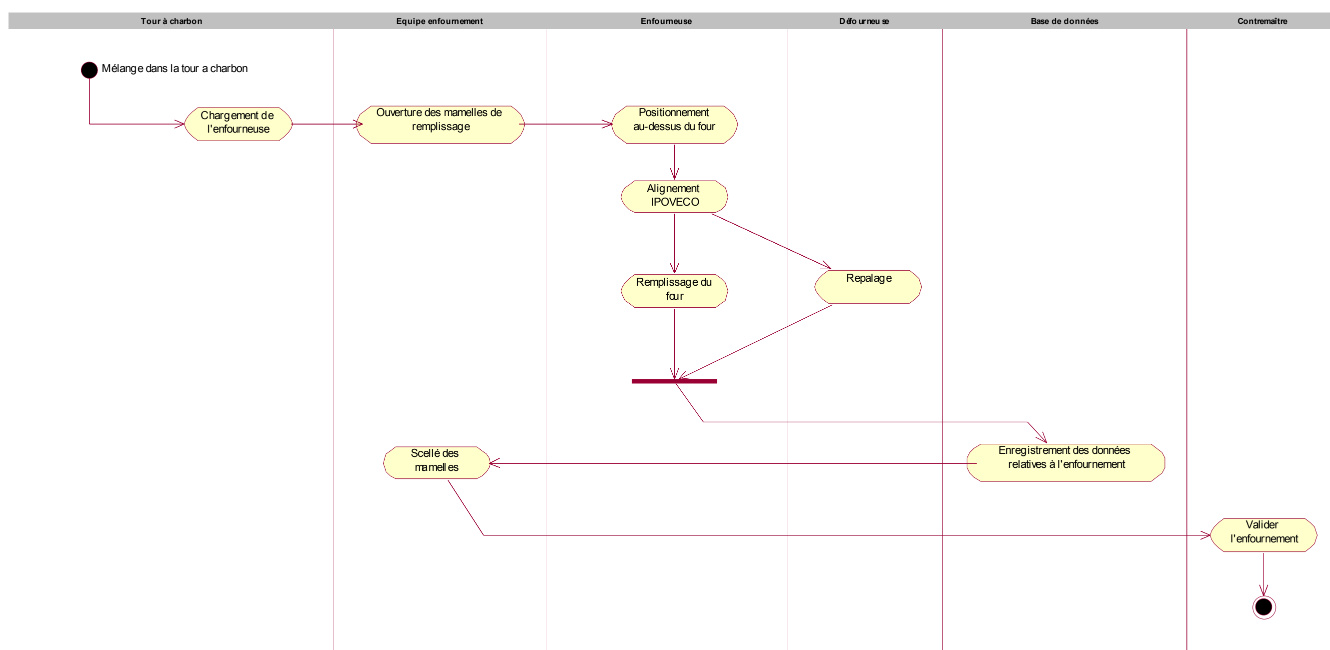

Figure 6 : Diagramme d'activités pour *Encoder température carneaux*

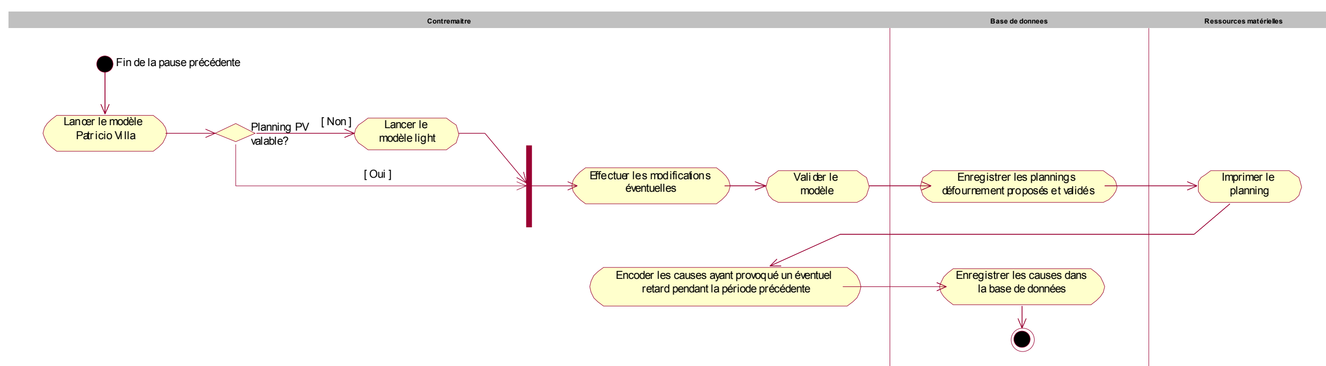

Figure 7 : Diagramme d'activités pour *Créer planning défournement*

Intervenant : le contremaître.
Cas d'utilisation associés : utiliser intranet, enfournement, défournement.

Encoder cahier électronique
Objectif : enregistrer les données relevées par l'équipe contrôle au cours d'une pause.
Description : l'équipe contrôle utilise l'intranet à la manière d'un cahier électronique, c'est-à-dire pour l'encodage d'une série d'informations comme des températures, des pressions, des vitesses, etc. relevées au cours d'une pause.
Intervenant : l'équipe contrôle.
Cas d'utilisation associé : utiliser intranet.

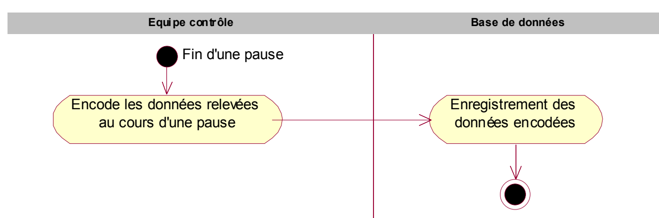

Figure 8 : Diagramme d'activités pour *Encoder cahier électronique*

Mélanger charbon
Objectif : obtenir la pâte à coke (matière prête à être enfournée).
Description : Carsid dispose de plusieurs charbons venant de différents endroits dans le monde et avec des propriétés différentes. Ces charbons sont mélangés et ensuite broyés pour obtenir de la pâte à coke. Le mélange est analysé et fait l'objet d'un rapport sur l'intranet.
Intervenant : équipe mélange, tapis roulant, broyeuse.
Cas d'utilisation associé : enfournement.

Enfournement
Objectif : remplir les fours de pâte à coke et relever des informations relatives à l'enfournement qui seront inscrites dans la base de données.
Description : lors d'un enfournement, l'enfourneuse *E3* fournit, par ses automatismes, des données de production : l'heure, le numéro du four, le poids et l'humidité de la pâte. Dans le cas d'un enfournement par l'enfourneuse *E1*, des valeurs par défaut sont utilisées.
Intervenant : tour à charbon, équipe enfournement, enfourneuse, défourneuse, contremaître.
Cas d'utilisation associé : cuisson, mélanger charbon.

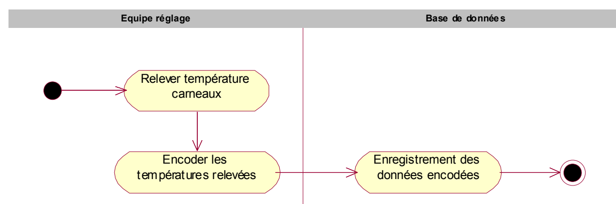

Figure 10. : Diagramme d'activités pour *Enfournement*

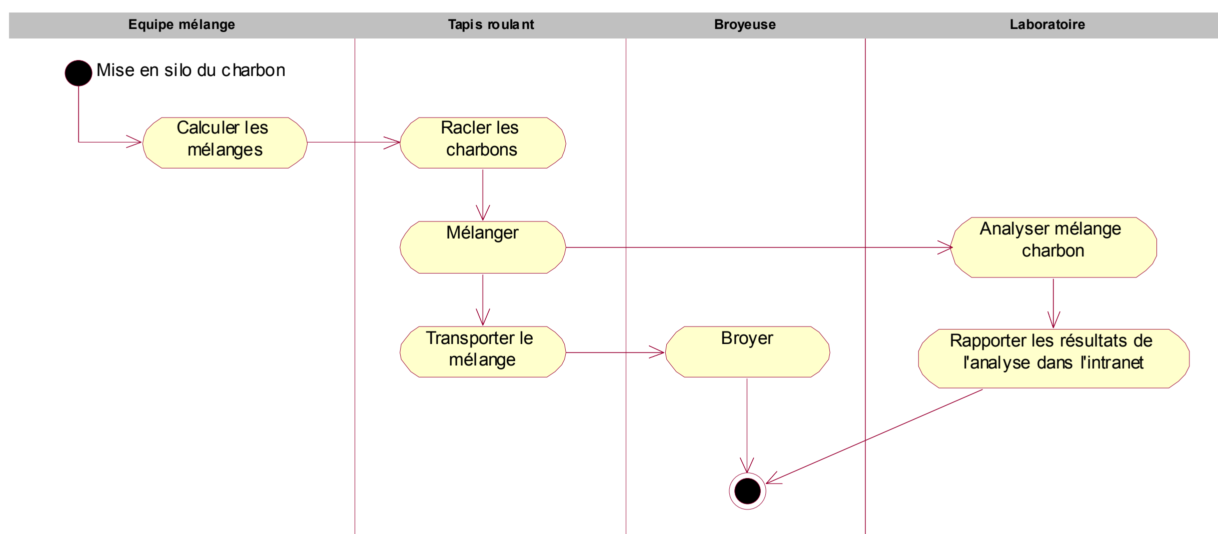

Figure 9 : diagramme d'activités pour *Mélanger charbons*

Cuisson
Objectif : transformer la pâte à coke en le saumon de coke.
Description : voir modélisation métier.
Intervenant : four.
Cas d'utilisation associé : enfournement, défournement, encoder température carneaux.

Défournement
Objectif : extraire le saumon de coke du four et le déverser dans le coke car.
Description : lors d'un défournement, une série d'informations sont enregistrées dans la base de données afin d'être exploitées par la suite par les décisionnaires. Il s'agit de la température (relevée par le système *Cotherm*), de la puissance de la défourneuse et du profil coke.
Intervenant : guide coke, défourneuse, coke car, four à coke, système Cotherm, contremaître.

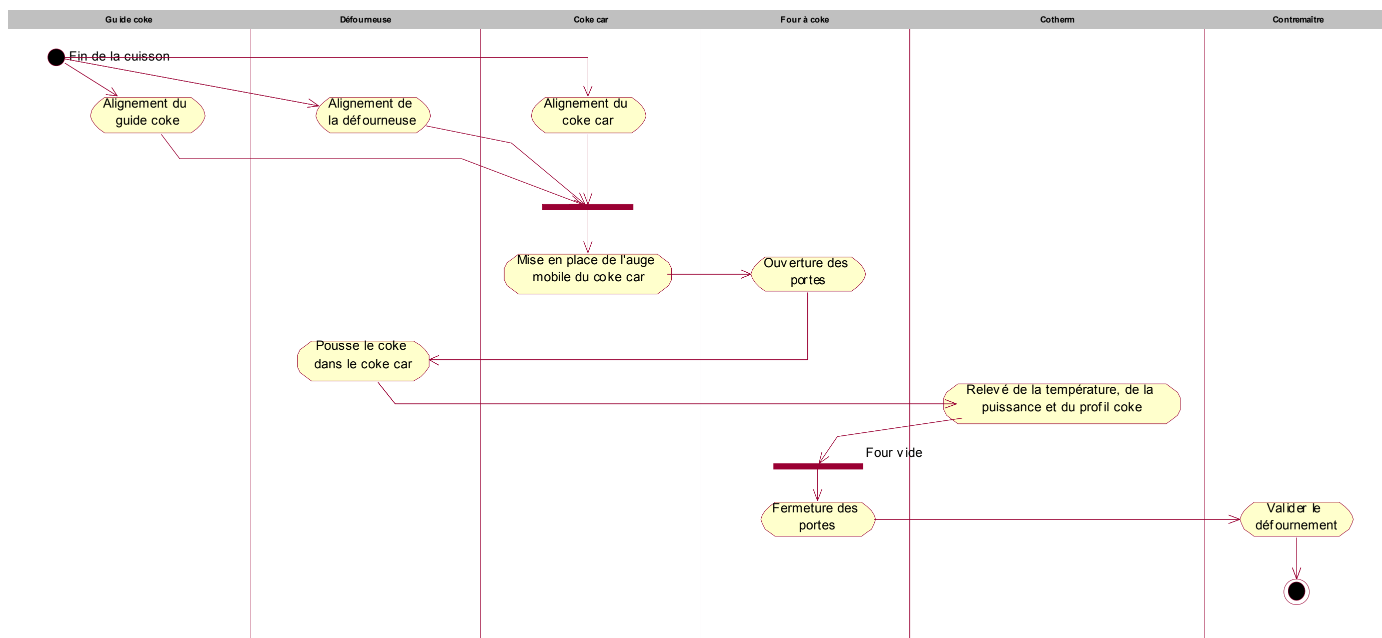

Figure 11 : diagramme d'activité pour *Défournement*

Figure 12 : Diagramme des cas d'utilisation cokerie

5 MODELISATION METIER DE LA TRACTION

De grandes quantités de matières entrent et sortent de l'entreprise Carsid chaque jour. Ces mouvements se font principalement de deux manières. La première est la traction, c'est-à-dire le transport par trains de marchandises qui est développé ci-dessous. La seconde est la bascule, c'est-à-dire le transport par camions, qui sera analysé ultérieurement dans ce document.
Les noms Traction et Bascule ont été donnés aux bases de données de Carsid qui concernent les mouvements de marchandise par voie ferrée et par la route. Ces deux bases de données seront intégrées à la base de données cokerie dans l'activité d'analyse et de conception.

5.1 Les concepts

Les descriptions qui suivent se basent sur des interviews d'Yves Wautelet et de Laurent Louvigny avec des membres du personnel de la cokerie, et avec des membres du service informatique de Carsid, ainsi que sur des documents internes sur le sujet. Une analyse de la traction réalisée par G. Lorge a également permis un travail plus en profondeur sur ce sujet.

L'exploitation de la traction. Le service de la traction est composé de trois secteurs. Le premier de ces secteurs est l'*Exploitation à la traction* qui a pour rôle de faire circuler la matière au bon moment entre les gares de livraison et les services de Carsid avec un minimum de locomotive. Elle est composée du dispatching, du bureau et du machiniste instructeur. C'est le dispatching qui gère en continu la position des locomotives de Carsid. Il dispose d'un système d'information local ayant une double fonction : la gestion des locomotives et des wagons en temps réel à l'aide d'un synoptique et l'acquisition des pesées des wagons. Le bureau quant à lui prépare et supervise les fonctions du dispatching. Finalement, le machiniste instructeur s'occupe du suivi du parc des locomotives (machines), il assure la coordination entre le dispatching et l'atelier de réparation du matériel roulant et il dirige aussi la formation des machinistes.

La gestion traction. La *Gestion traction* est le deuxième secteur. Elle s'occupe de la gestion des convois en relation avec la SNCB, notamment de la rédaction des lettres de voitures et des bordereaux de réception. Elle gère aussi la préfacturation des transports externes : minerais, charbons, ferrailles, brames, etc.
C'est elle aussi qui s'occupe de la commande des wagons, et qui gère les avaries et les chômages, ainsi que le traitement des wagons de mitraille.

Les autres services. D'*autres services* appartiennent au Service traction, et on peut considérer qu'ils forment un troisième secteur. Par exemple, il existe un atelier central disposant des équipes pour l'entretien et la réparation des wagons, des locomotives et des voies. Un service est également chargé de réceptionner la mitraille et de nettoyer les wagons qui la transportent.

Les mouvements internes et externes. Les *mouvements internes* sont effectués entre deux points du site de l'entreprise Carsid afin de transporter diverses matières au sein même de l'entreprise. Pour ce faire, le dispatching forme un train et ordonne un mouvement de celui-ci vers l'endroit où se trouvent les marchandises à transporter puis, une fois chargé, ce train est transporté vers son lieu de déchargement.
Les *mouvements externes* quant à eux sont destinés à faire entrer ou sortir de la matière de l'entreprise. Cependant, les locomotives de Carsid ne sillonnent pas la Belgique pour acheminer la matière première et livrer les produits de l'entreprise. Elles se contentent d'amener les wagons qui doivent sortir de Carsid à quelques kilomètres de là, sur l'une des voies mises à disposition par la SNCB. Les convois y attendent d'être pris en charge par une locomotive de la SNCB qui accomplira le reste du parcours. C'est aussi là qu'arrivent les wagons provenant de l'extérieur et que les locomotives de Carsid viendront chercher pour les emmener sur le site.

Marcinelle, Anvers et Sandréa. Le dispatching de Carsid, situé à *Marcinelle*, ne s'occupe pas seulement des mouvements internes et externes propres au site de Marcinelle, mais également des mouvements de trains au niveau du site d'*Anvers* et du stock *Sandréa*. Le port d'Anvers est un passage incontournable pour nombre de marchandise, et le monde de la sidérurgie ne fait pas exception. Comme il n'y a pas de personnel administratif au port d'Anvers, on le considère comme un site fictif de Carsid afin de pouvoir générer une *lettre de*

voiture électronique. De même, le stock Sandréa est une plate-forme multimodale (fer, eau, route) qui sert de tampon entre les fournisseurs de ferrailles et Carsid. Il y a très peu de personnel sur ce site, et il n'y pas d'équipement électronique pour produire la lettre de voiture Celle-ci est donc prise en charge par Carsid.

Les voies S. Les *voie S* sont des voies fictives qui ne sont pas physiquement sur le site de Carsid mais en font néanmoins « informatiquement » partie. Originellement la lettre « S » provient de SNCB mais l'appellation voie S est aussi utilisée pour les voies situées sur d'autres sites privés comme Sandréa, un sous-traitant situé à Charleroi ou ABT à Anvers, avec lesquels Carsid a des relations privilégiées. A proximité du site de Marcinelle se trouvent vingt-cinq voies dites *voies S*, et à proximité du site d'Anvers il y en a six.
Pour donner un exemple de l'utilisation des voies S, supposons qu'un convoi de charbon doit être transporté depuis Anvers vers Marcinelle. Le dispatching commence par donner l'ordre de mettre le train sur l'une des six voies S disponibles à Anvers et envoie un télex à la SNCB lui signifiant de transporter le convoi vers sa destination, c'est-à-dire l'une des vingt-cinq voies S de Marcinelle. Notons que le train possède un numéro interne (propre à Carsid) différent du numéro externe (propre à la SNCB). Une fois le train arrivé à Marcinelle, la SNCB envoie un télex au dispatching, et le train attend sur la voie S d'être pris en charge par une locomotive de Carsid. C'est au dispatching de déterminer le moment adéquat, en fonction des priorités, pour déplacer le convoi de la voie S vers son lieu d'acheminement dans l'usine.

La SNCB. La plupart des wagons sur le site de Carsid appartiennent à la *SNCB*, à l'exception des poches à fonte et de quelques autres qui sont appelés « wagons particuliers ». Par contre les locomotives sur le site sont la propriété de Carsid. Elles effectuent les mouvements internes, et elles acheminent les wagons désignés sur le réseau SNCB afin que des locomotives de la SNCB viennent les prendre en charge pour effectuer un mouvement externe.
Pour être plus précis, le transport des marchandises est assuré par B_Cargo, filiale à 100% de la SNCB. Deux types de gares lui sont spécifiques : les gares de triage où les wagons d'une rame de marchandises peuvent être séparés pour être redirigés et les gares de livraison qui sont des zones tampons entre la SNCB et le client.

La lettre de voiture. En règle générale, lorsqu'il y a transport de marchandises entre une entreprise expéditrice et une entreprise destinataire, il y a création d'une *lettre de voiture*. C'est par le biais de la lettre de voiture (papier et/ou électronique) que l'expéditeur communique à la SNCB l'adresse de destination, la liste des wagons à emmener et leur contenu. La SNCB souhaite que la lettre de voiture soit produite informatiquement par l'expéditeur afin de réduire le nombre de litiges avec ses clients.
C'est sur base des éléments contenus dans la lettre de voiture que la facture relative au transport est calculée. C'est la raison pour laquelle on y trouve notamment le nom de l'expéditeur, du destinataire, un numéro de contrat SNCB et un code TVA.
Dans le cas du trafic intérieur en Belgique, une version simplifiée de la lettre de voiture peut être utilisée, on l'appelle « ordre de transport ».
Carsid n'est cependant pas tenu de rédiger une lettre de voiture lors de la restitution de wagons vides appartenant à la SNCB. Celle-ci repère en effet visuellement tout passage de wagon en gare, et constate d'elle-même la restitution de ses wagons.

Les telex. Pour chaque mouvement de rame concernant Carsid sur le réseau SNCB, le dispatching reçoit un message qui est imprimé sous forme d'un *télex*.
Il existe différent types de télex :

- restitution : les wagons pleins ou vides sont restitués à la SNCB.
- annonce départ : les wagons pleins quittent la gare de l'expéditeur.
- annonce d'arrivée : les wagons pleins arrivent en gare de destination.
- arrivée à destination : les wagons pleins rentrent chez le destinataire.
- mise à disposition : la SNCB met ses wagons vides à la disposition de son client.

Le chômage. Les wagons utilisés pour le transport de marchandises mais appartenant à la SNCB ne peuvent rester sur le site de Marcinelle qu'un temps déterminé. En effet, une fois que ces wagons sont délivrés par le transporteur, ils entrent dans une phase, dite de *chômage*, calculée en nombre d'heures. Ce temps de chômage ne peut dépasser 22 heures pour le chargement ou le déchargement sous peine d'une amende horaire à dater de la 23e heure de chômage. C'est à la Gestion traction que revient le rôle de gérer le chômage des wagons sachant qu'en fonction de la matière transportée, les wagons

doivent subir des traitements différents. En effet, les wagons transportant des mitrailles doivent être pesés avant d'être déchargés sur le site de Marcinelle et pour ce faire, ils doivent passer sur un type spécial de bascule pour wagons. Avant de pouvoir être rendus à la SNCB, les wagons de mitrailles doivent également être nettoyés par le service de nettoyage. Dès lors, et contrairement aux autres wagons, les wagons de mitrailles ne peuvent être simplement déchargés ou chargés puis transportés à nouveau vers l'extérieur.
Notons qu'il n'y a pas de chômage ni le dimanche ni les jours fériés, et que deux heures de chômage supplémentaires sont allouées pour le nettoyage des wagons à ferrailles. Pour donner un exemple, si on considère un wagon de mitraille devant être déchargé, nettoyé et chargé à nouveau, le temps de chômage maximum avant de devoir payer une amende à la SNCB est de 46h (22h + 2h + 22h).
En fin de mois, la SNCB adresse à la Gestion traction le décompte global du séjour des wagons sur raccordement, par raccordement et par catégorie de wagons.
Les frais de chômage relatifs aux wagons de ferrailles se situent entre 50.000 et 70.000 EUR par mois. Cependant, le coût du chômage pour l'ensemble des wagons n'est que d'environ 5.000 EUR par mois car les wagons à minerais, charbons et ferrailles sont actuellement compris dans une même catégorie par la SNCB, et un décompte global des heures de chômage est possible. Carsid compense donc les heures de chômage en trop des wagons à ferrailles par la rotation rapide des wagons à charbons et minerais qui ne restent en moyenne que quelques heures dans l'usine.
Malheureusement, dans les prochains mois, la SNCB va revoir les catégories et les compensations ne seront plus d'actualité.

Les avaries. Lors de la restitution des wagons (pleins ou vides) en gare, des visiteurs SNCB viennent contrôler l'état de tout matériel roulant avant de le laisser partir vers sa destination. En cas de problème, le wagon victime d'une *avarie* est déplacé sur une voie particulière pour réparations. Celles-ci sont effectuées par la SNCB qui facture les frais à l'expéditeur.
Les avaries sont répertoriées, chacune d'elles a un coût standard comprenant matière et main d'œuvre. Le visiteur peut constater plusieurs avaries sur un même wagon. On constate statistiquement que 70 à 80% des avaries concernent les wagons à ferrailles, très peu les wagons à minerais et charbons.
Chaque jour, la SNCB transmet à CARSID un procès verbal manuscrit qui décrit la nature des avaries par wagon avarié. La Gestion traction valide ou non ce document puis l'encode dans le système informatique. En fin de mois, la SNCB transmet une proposition de facture à laquelle est jointe une « annexe à la facture ». Celle-ci sert en même temps de devis de réparations. Elle détaille par avarie le coût de chaque poste pour tous les wagons du mois écoulé. La Gestion traction complète l'information partielle, introduite quotidiennement, par les prix communiqués dans le devis pour produire la *préfacture* de Carsid.
Puis elle compare le devis de la SNCB avec cette préfacture. Après accord entre les deux parties, la Gestion traction introduit une demande d'achat de régularisation et fournit à la comptabilité analytique les proportions des dépenses par service.
Il arrive couramment qu'une facture reprenne des avaries causées depuis des mois et oubliées à l'époque par la SNCB. Ceci explique qu'il faut garder un certain temps l'historique des avaries.

5.2 La modélisation

Pour traiter la modélisation métier de la traction, un diagramme des cas d'utilisation à été définit. L'enchaînement d'activités au sein de chaque cas d'utilisation est représenté par les diagrammes d'activités présentés ci-dessous.

Gérer les avaries
Objectif : dédommager la SNCB si un de ses wagons a été endommagé.
Description : lorsqu'un wagon ayant transité par Carsid est endommagé, la SNCB rédige un procès-verbal d'avarie qui parvient ensuite à la Gestion traction. En fin de mois la SNCB envoie une proposition de facture comprenant les devis pour chaque avarie. La Gestion traction rédige alors sa propre préfacture, la compare avec la proposition de facture SNCB, et paie cette dernière.
Intervenants : SNCB, Gestion traction.

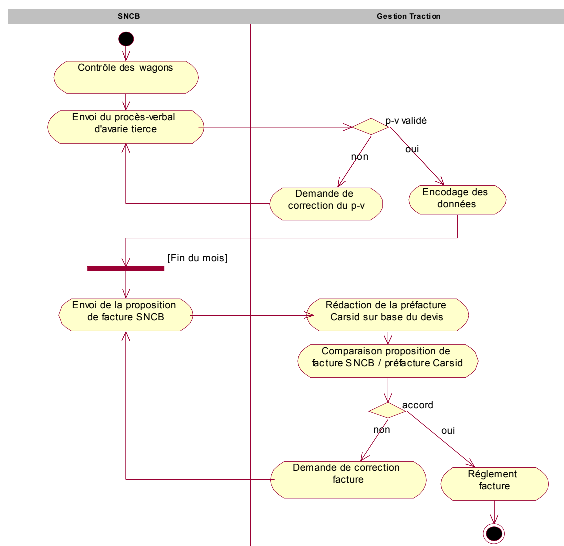

Figure 13 : diagramme d'activité pour *Gérer les avaries*

Encoder lettre de voiture.
Objectif : rédiger un contrat de transport entre l'expéditeur et le destinataire.
Description : lorsqu'il y a transport de marchandises, la Gestion traction crée une lettre de voiture.
Intervenants : Gestion traction, SNCB.

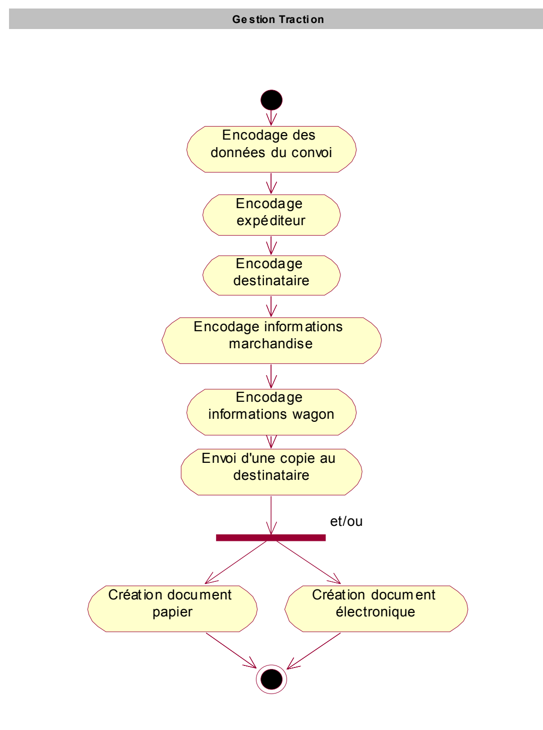

Figure 14 : diagramme d'activité pour *Encoder lettre de voiture*

Gérer les chômages
Objectif : éviter les temps de chômage excessifs et payer l'amende mensuelle des heures de chômage à la SNCB.
Description : La Gestion traction doit agréger les heures de chômage sur base mensuelle et payer l'amende éventuelle à la SNCB.
Intervenants : SNCB, Gestion traction.
Cas d'utilisation associé : Traiter les wagons de mitraille.

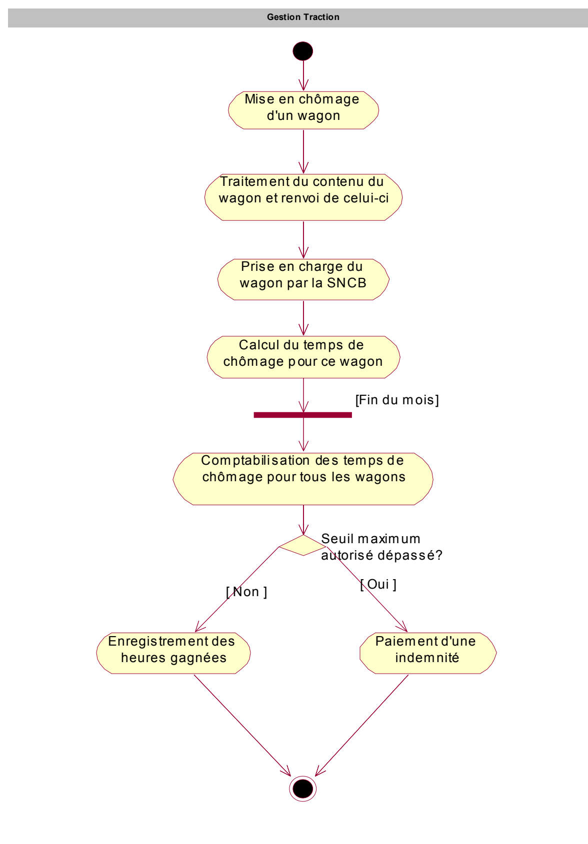

Figure 15 : diagramme d'activité pour *Gérer les chômages*

Effectuer mouvement interne
Objectif : déplacer un convoi de wagons d'un point à un autre du site de Marcinelle.
Description : le dispatching donne l'ordre de déplacer un train d'un point à un autre, le déplacement à lieu et le dispatching l'encode sur le synoptique.
Intervenant : dispatching.
Cas d'utilisation associé : Traiter les wagons de mitraille.

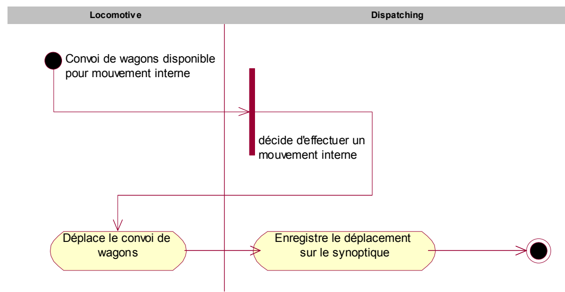

Figure 16 : diagramme d'activité pour *Effectuer mouvement interne*

Effectuer mouvement externe
Objectif : déplacer un convoi de wagons vers un autre site.
Description : un convoi de wagons doit être acheminé par le transporteur extérieur (la SNCB) d'un site à l'autre. Pour ce faire, le convoi doit être placé sur la *voie S* du site d'origine et un télex doit être envoyé à la SNCB pour que celle-ci vienne le chercher. Une fois le convoi arrivé à destination, il est placé sur la voie S de ce site et un télex est envoyé au dispatching par la SNCB. La Gestion traction doit enregistrer le début du chômage si le mouvement externe correspond à une arrivée d'un convoi sur le site de Marcinelle et enregistrer la fin du temps de chômage si le mouvement externe est un départ depuis ce site.
Intervenants : SNCB, dispatching, Gestion traction.

Cas d'utilisation associé : Traiter les wagons de mitraille.

Associer n° train
Objectif : associer un numéro de train interne à un convoi qui a été créé.
Description : lorsqu'un train est formé, on doit lui associer un numéro interne (différent du numéro qu'il porte au sein de la SNCB). Ce numéro encodé sur le synoptique lui permet d'être identifié dans la base de données.
Intervenant : dispatching.

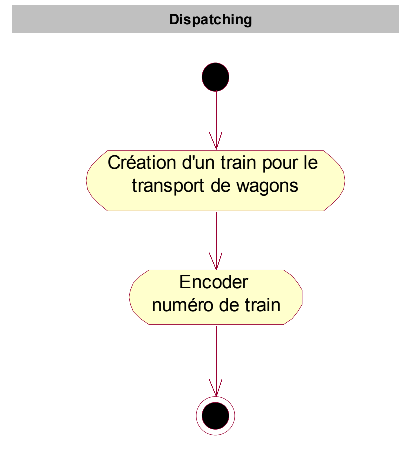

Figure 18 : diagramme d'activité pour *Associer n° de train*

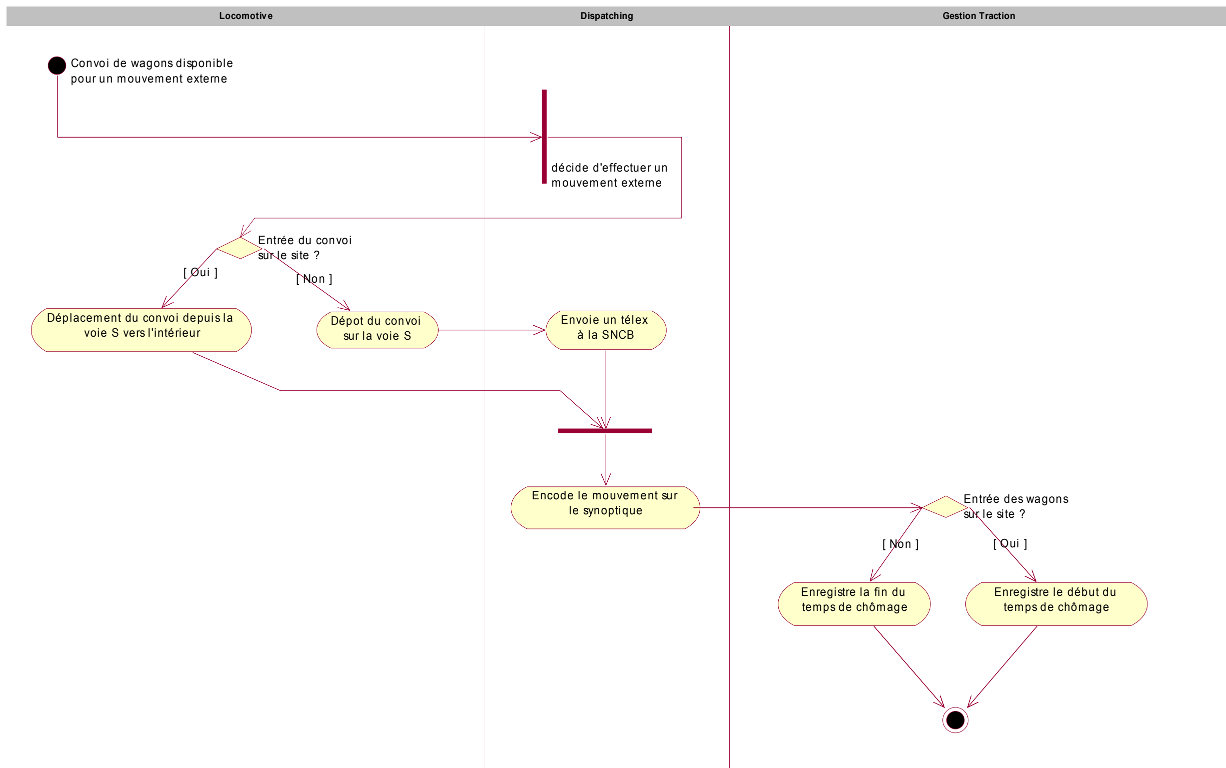

Figure 17 : Diagramme d'activité pour *Effectuer mouvement externe*

Traiter les wagons de mitraille

Objectif : peser, décharger et nettoyer les wagons de mitraille.

Description : contrairement aux autres types de wagons qui sont pesés à Anvers, les wagons de mitraille doivent être pesés à Marcinelle pour que le service comptable puisse déterminer la quantité de mitraille à payer au ferrailleur. Le wagon de mitraille passe donc sur la bascule wagon avant être nettoyés par le service nettoyage avant d'être rendus à la SNCB.

Intervenants : SNCB, Gestion traction, dispatching, bascule wagons, réceptionniste mitraille, service nettoyage.

Cas d'utilisation associés : Gérer les chômages, Effectuer mouvement externe, Effectuer mouvement interne.

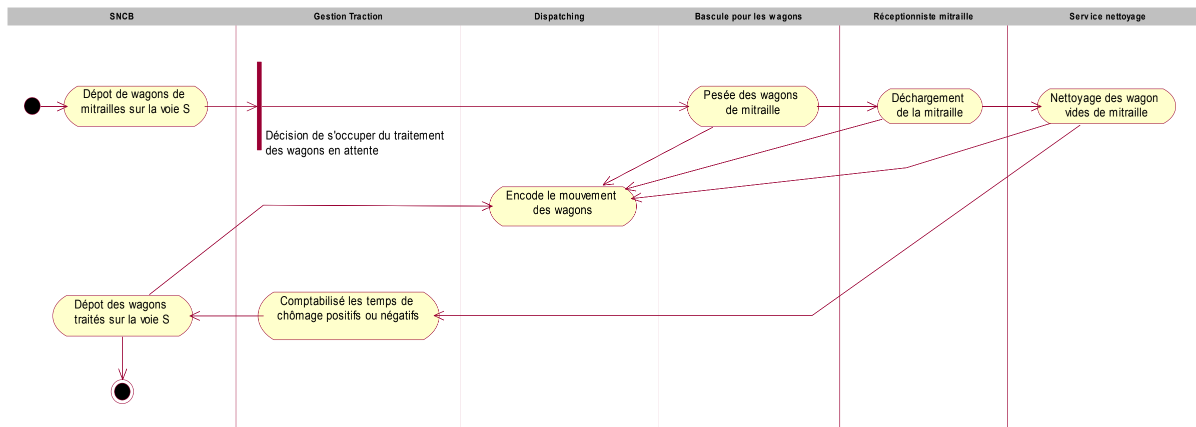

Figure 19 : diagramme d'activité pour *Traiter des wagons de mitraille*

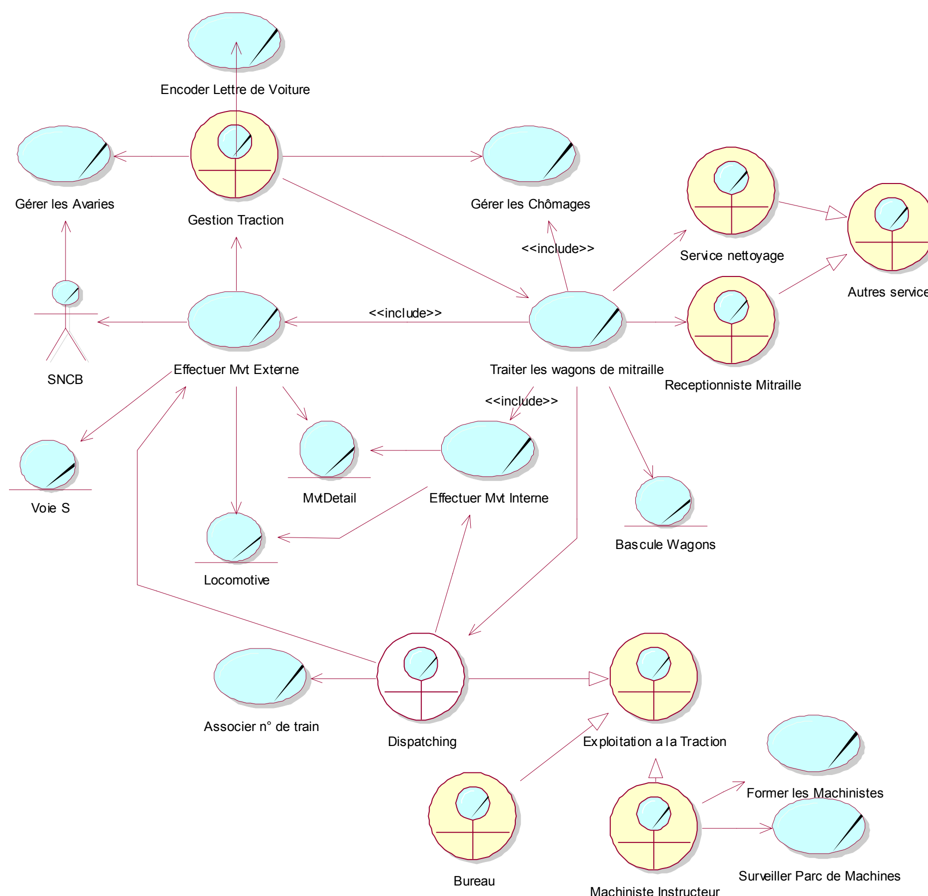

Figure 20 : Diagramme des cas d'utilisation traction

6 MODELISATION METIER DE LA BASCULE

La route constitue une voie privilégiée pour les flux de matière entrant et sortant de chez Carsid. Le mot « bascule » fait initialement référence à la machine qui permet de peser les camions qui entrent et qui sortent de chez Carsid, mais ce terme désigne également la base de données récoltant les informations concernant les transport de matières par camion.
En comparant la masse à vide et la masse à charge d'un camion, la bascule permet de déterminer la masse totale des matières transportées (apportées par le fournisseur ou transportées vers le client). Cela permet à Carsid de savoir exactement quelle quantité de produit lui sera facturée ou quelle quantité facturer à ses clients.

6.1 Les concepts

Le badge
Un *badge* est un identifiant auquel sont associées toutes les données concernant un camion entrant dans l'usine Carsid (nom du fournisseur, n° de plaque, nom du transporteur,etc.). L'utilisation du badge peut être unique ou bien servir lors de plusieurs transports. Il y a trois types de badge :

- occasionnel, livraison unique « in » ou « out » ;
- permanent, un badge pour toutes les pesées d'une seule commande ;
- tournant, tarage du camion tous les x transports.

Lorsqu'un camion arrive devant les portes de Carsid deux situations sont possible. Premièrement, le chauffeur n'a pas de badge pour ce camion, il en reçoit alors un et pèse son camion sur la bascule. Au moment de la pesée, il introduit le badge dans une borne puis le récupère. Il va ensuite charger ou décharger ses marchandises, puis revient se peser sur la bascule. C'est la différence des deux pesées qui permet de calculer le poids des marchandises transportées.
Dans un deuxième cas de figure, le chauffeur a déjà un badge associé au camion. S'il arrive à vide, la pesée n'est cette fois plus nécessaire. Il va charger les marchandises, et seulement à ce moment pèse son camion pour déterminer le poids des marchandises. S'il arrive chargé, il commence au contraire par peser son camion, décharge les marchandises, et peut alors partir sans besoin d'une autre pesée.
Notons que le badge est avalé automatiquement par la borne de la bascule lors de sa dernière utilisation.

Le basculeur
Le *basculeur* est un employé qui se trouve aux entrées de Carsid. Ses rôles principaux sont l'attribution de badges aux camionneurs qui doivent entrer sur le site, ainsi que la surveillance des fraudes possibles concernant les pesées.

Le camionneur
Le *camionneur* par contre travaille pour un transporteur indépendant de Carsid, et a pour mission de transporter les marchandises assignées.
Voyons un exemple de fraudes que le basculeur doit empêcher.Supposons qu'un camionneur doit livrer une certaine quantité de minerai de fer à Carsid, en faisant pour cela plusieurs trajets. Il peut essayer de tricher sur la quantité réellement livrée en utilisant deux camions différents lors du transport du minerai. Lors de la première livraison, il utilisera un camion léger dont le poids à vide sera enregistré par Carsid.
En utilisant lors des livraisons ultérieures un camion plus lourd, tout en utilisant le badge du premier camion, il pourra tricher sur la quantité de minerai qu'il apporte réellement. En effet, le poids de la livraison sera calculé en soustrayant du poids à charge du camion lourd le poids à vide du camion plus léger, gagnant ainsi à chaque livraison la différence de poids entre les deux camions.

Le ticket
A partir de la deuxième pesée, le chauffeur reçoit à chaque pesée deux *tickets* identiques délivrés par la borne où il introduit le badge et comportant certaines informations (date, heure, poids à charge, poids à vide, etc.). Dans la plupart des cas, il garde un ticket et va porter l'autre chez le basculeur. Ce dernier ticket est alors envoyé au service comptable.

Les principaux concepts relatifs à la bascule ayant été mis en place, penchons-nous à présent sur les résultats de notre analyse.

6.2 La modélisation

Pour traiter la modélisation métier de la traction, un diagramme des cas d'utilisation à été définit. L'enchaînement d'activités au sein de chaque cas d'utilisation est représenté par les diagrammes d'activités présentés ci-dessous.

Attribuer un badge
Objectif : attribuer un badge valide à un camion pour qu'il puisse effectuer au moins une pesée.
Description : lorsqu'un camion se présente à l'usine et qu'il n'a pas de badge valide, il doit se présenter chez le basculeur qui lui attribue un badge en fonction des informations qu'il lui donne. Le camionneur est ensuite à même d'effectuer la pesée.
Intervenants : basculeur, camion.
Cas d'utilisation associé : peser un véhicule.

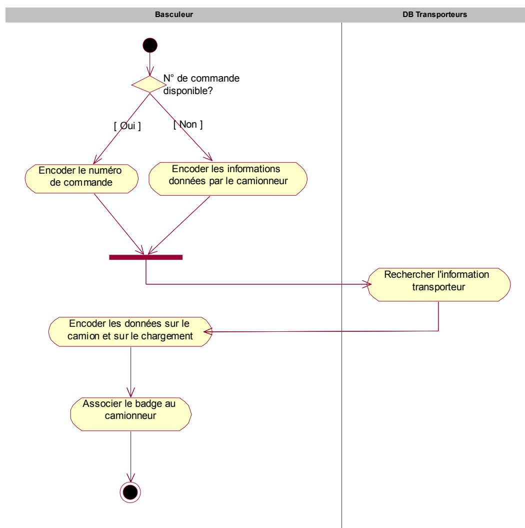

Figure 21 : Diagramme d'activité pour *Attribuer un badge*

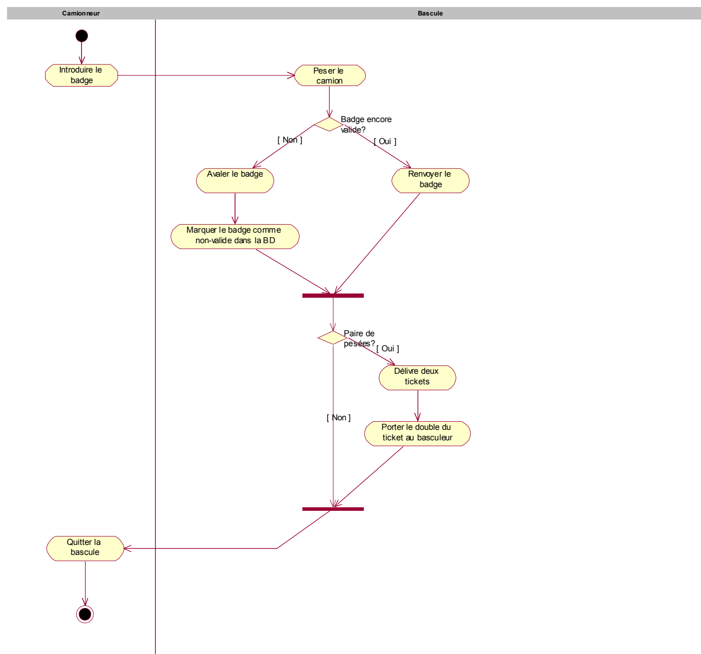

Figure 22 : Diagramme d'activité pour *Peser du véhicule*

Peser un véhicule
Objectif : déterminer la masse à vide ou à plein du véhicule.
Description : un camion muni d'un badge valide se présente sur la bascule, y introduit son badge et se fait peser soit à plein et à vide dans le cas d'une livraison ou soit à vide et à plein dans le cas d'un retrait de marchandises.
Intervenants : bascule, camion.
Cas d'utilisation associés : livrer, attribuer un badge.

Livrer
Objectif : vider ou remplir le camion après la première pesée.
Description : une fois pesé, le camion va aux stocks charger ou décharger une cargaison. S'il livre de la mitraille, il se rend chez le spécialiste mitraille.
Intervenants : camion, réceptionniste mitraille.
Cas d'utilisation associés : charger, décharger, peser un véhicule.

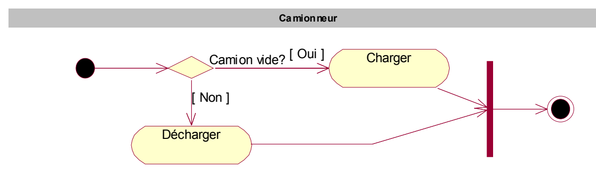

Figure 23 : Diagramme d'activité pour *Livrer*

Surveiller un camion
Objectif : vérifier que le camionneur n'essaye pas de frauder de quelque façon que ce soit.
Description : une fois le badge délivré, le camionneur peut aller peser son camion à vide et à plein. Lors de ces opérations, il peut tenter de frauder en effectuant la livraison ou le plein de marchandises en deux fois ou encore en utilisant des camions différents lorsqu'il utilise un badge à pesées multiples. Le basculeur est chargé de surveiller ce type de fraude et de bannir à tout jamais le transporteur en charge du camion pris à frauder.
Intervenants : basculeur, camion.

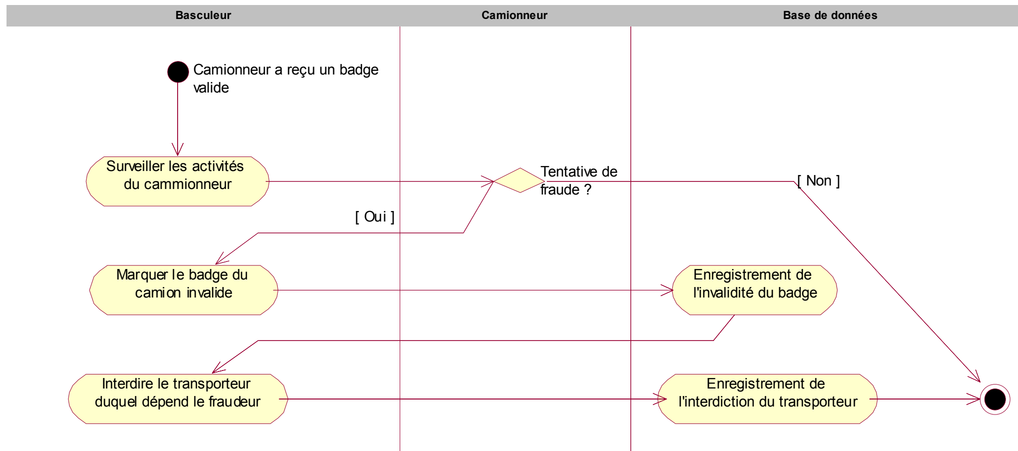

Figure 24 : Diagramme d'activité pour *Surveiller un camion*

surveillance du camion
surveiller un camion
Basculeur
attribution d'un badge
attribuer un badge
Camion
Receptionniste Mitraille
<<include>>
Badge
peser un véhicule
<<include>>
livrer
livraison
Charger
Décharger
Bascule
pesée d'un véhicule

Figure 25 : Diagramme des cas d'utilisation bascule

7 ANALYSE ET CONCEPTION

7.1 Méthodologie

Les travaux de rétro-ingénierie des bases de données existantes réalisés par Marti Ibarz Roger à la fin de l'année 2002 ont conduit à un diagramme de classe représentant l'architecture du système à l'étude. Les méthodes définies suite aux modélisations réalisées et présentées dans ce document ont été intégrées dans les classes de ce diagramme. Cette approche à été choisie dans le but de recycler les bases de données existantes.

7.2 Méthodes de la cokerie

Les premières activités de modélisation (modélisation métier et gestion des exigences) ayant été effectuées et l'architecture du système étant définie, il est maintenant impératif de passer à l'analyse du système, c'est-à-dire préciser les spécifications formelles des méthodes utiles au système. Les diagrammes d'activité ont permis de dégager les méthodes utiles pour satisfaire les fonctionnalités définies. La dynamique de ces méthodes peut se visualiser dans les diagrammes de séquence qui suivent. La spécification formelle de chaque méthode est détaillée en annexe. « *Une spécification est vue comme un contrat entre les implémenteurs d'une méthode et ses utilisateurs. Les implémenteurs s'engagent à réaliser la méthode conformément au contrat et les utilisateurs s'engagent à faire appel à la méthode sans supposer quoi que ce soit en dehors du contrat explicitement spécifié. Cet aspect constitue une des conditions nécessaires pour la réussite d'un développement modulaire d'un logiciel de taille importante.* »
En vue d'écrire des spécifications compréhensibles par tous, il a été fait usage de préconditions (PRE) à l'entrée de la méthode et de postconditions (POST) à la sortie de celle-ci. « *Les préconditions sont les assertions, c'est-à-dire les propositions ou prédicats qui sont supposés vrais, au moment de l'appel à la méthode à laquelle elles sont attachées. Ces assertions décrivent les états initiaux, les propriétés sur les entrées qui doivent être satisfaites avant l'exécution de la méthode pour que celle-ci produise les résultats attendus. Les préconditions peuvent porter sur les valeurs des paramètres passées à la méthode, sur l'état de l'objet courant représenté par les valeurs des variables d'instance ou, plus généralement, sur toute information pouvant influer sur son exécution.* »
« *Les postconditions sont les propriétés qui doivent être satisfaites à la fin de l'exécution de la méthode. Les postconditions décrivent notamment les propriétés que doit satisfaire le résultat de la méthode. Elles peuvent porter sur les valeurs renvoyées par la méthode, sur les exceptions lancées dans certains cas précisément définis, ou sur toute modification apportée lors de l'exécution de la méthode sur l'état de l'objet courant et, plus généralement, sur toute information modifiée lors de l'exécution de la méthode.* »
Le diagramme de classe complet de l'application est présenté dans les Figures 38 et 39.

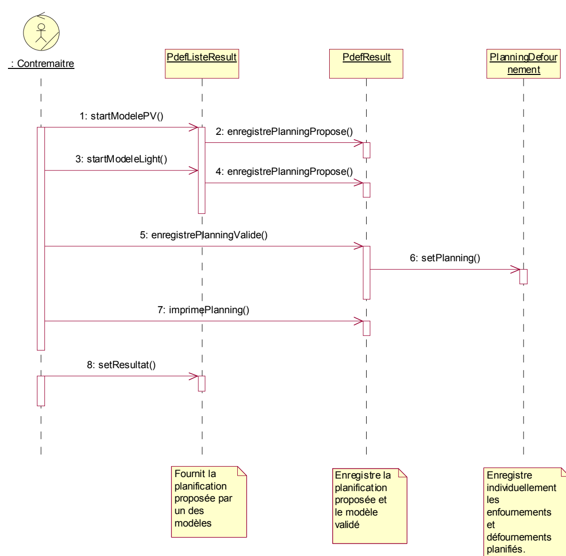

Figure 26 : Diagramme de séquence *Créer planning défournement*

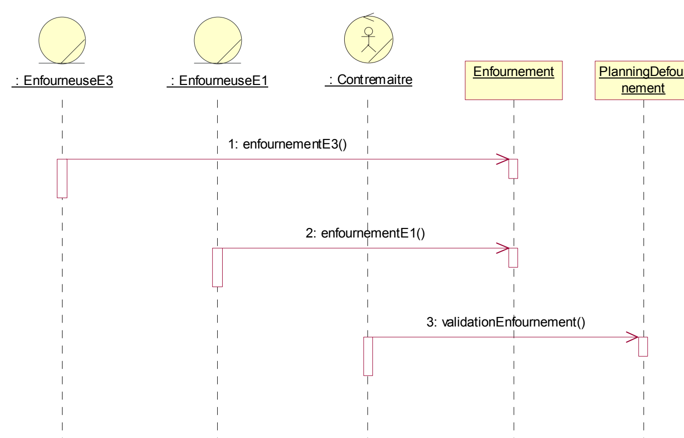

Figure 27 : Diagramme de séquence *Enfournement*

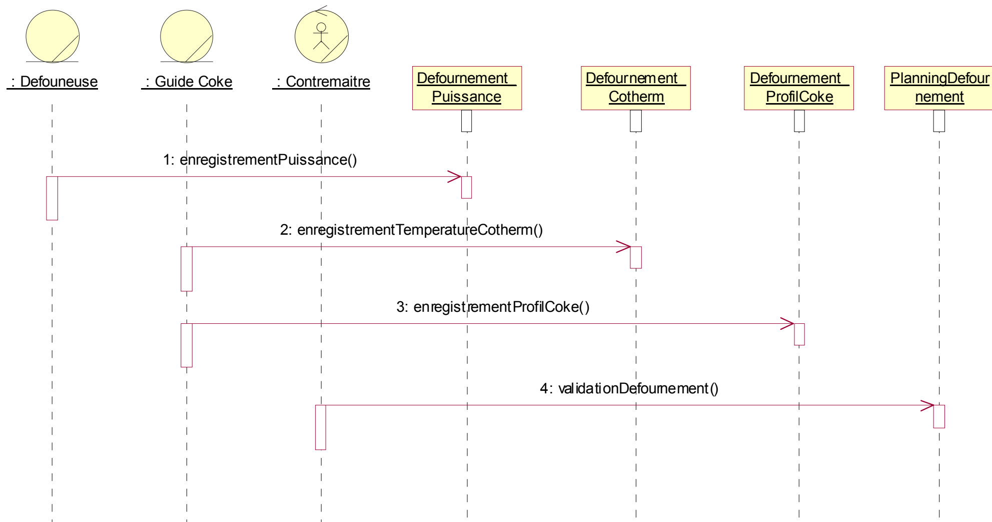

Figure 28 : Diagramme de séquence *Défournement*

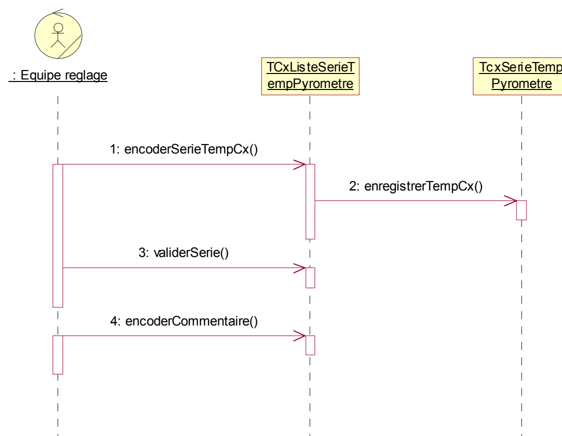

Figure 29 : Diagramme de séquence *Encoder température carneau*

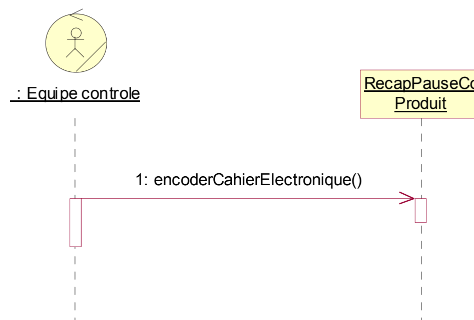

Figure 30 : Diagramme de séquence *Encoder cahier électronique*

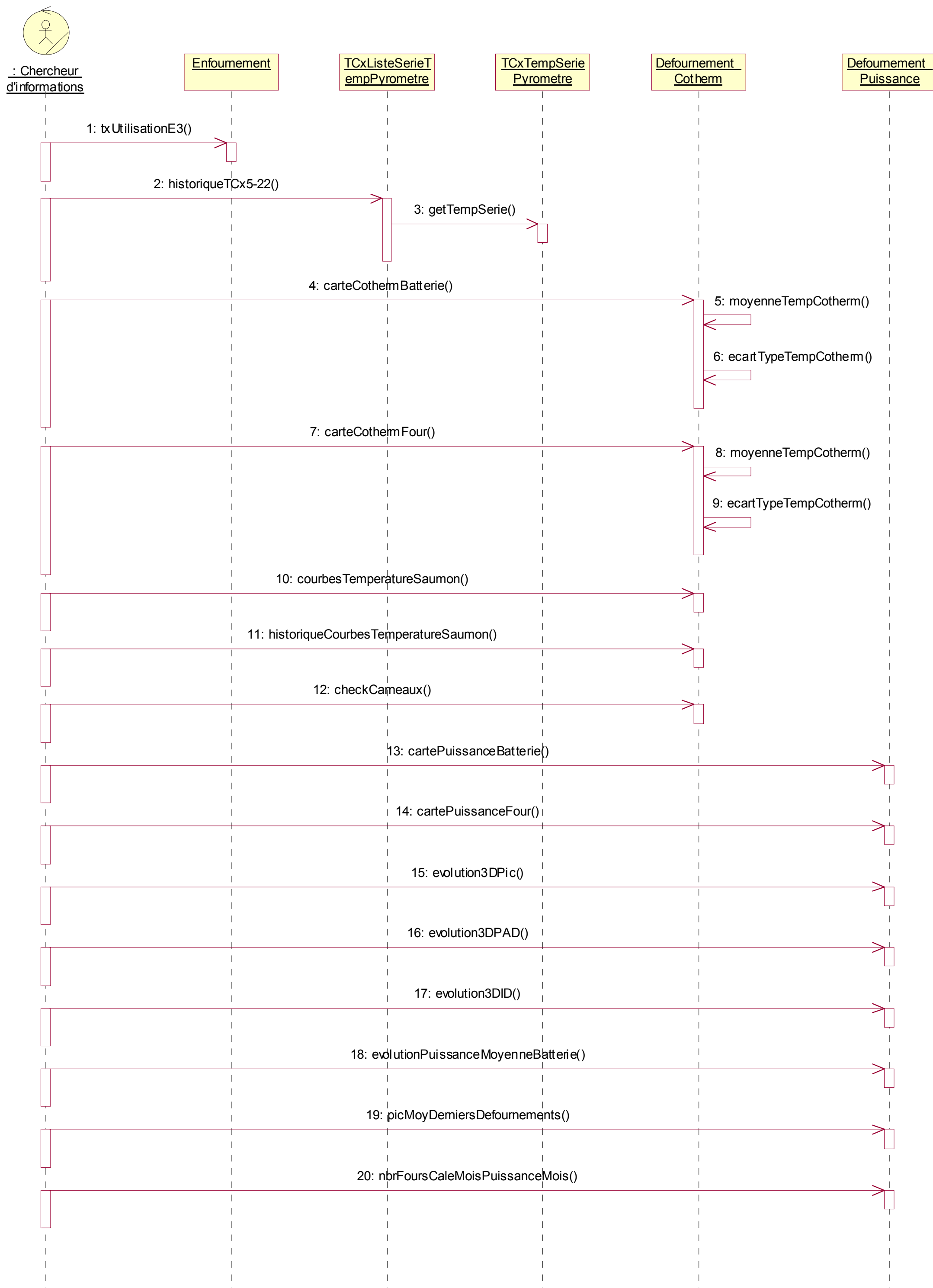

Figure 31 : Diagramme de séquence *Rechercher information*

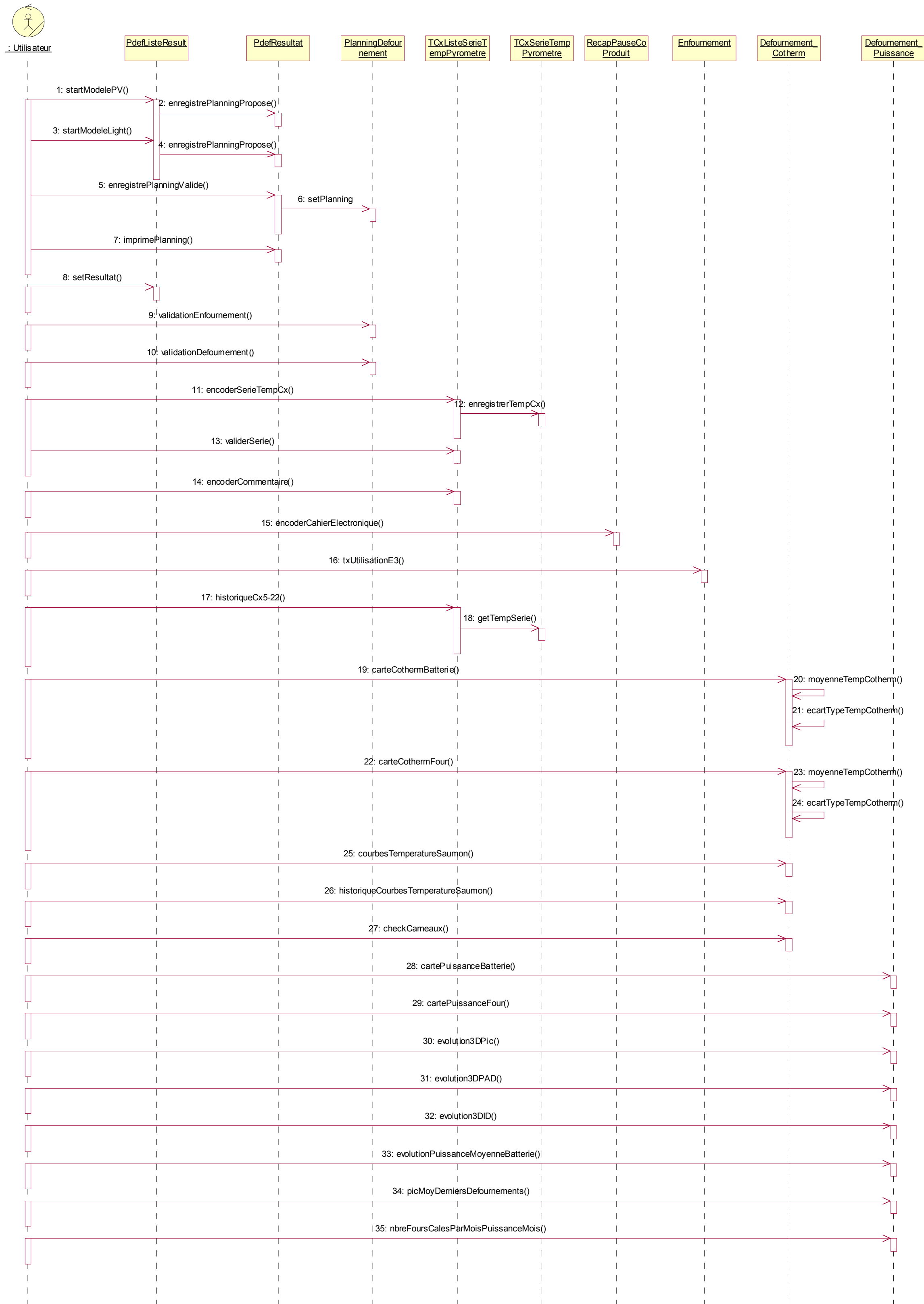

Figure 32 : Diagramme de séquence *Utiliser intranet*

7.3 Méthodes de la traction

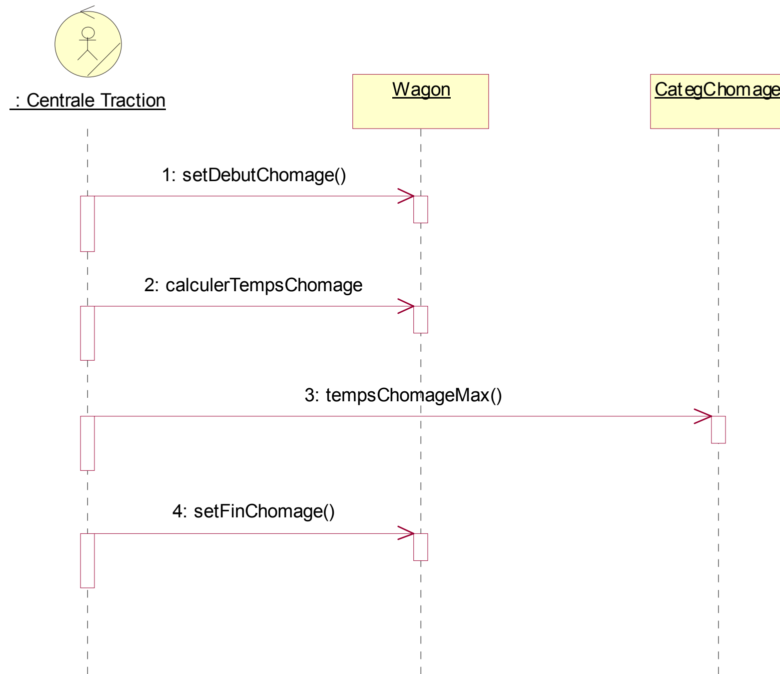

Figure 33 : Diagramme de séquence *Gérer les chômages*

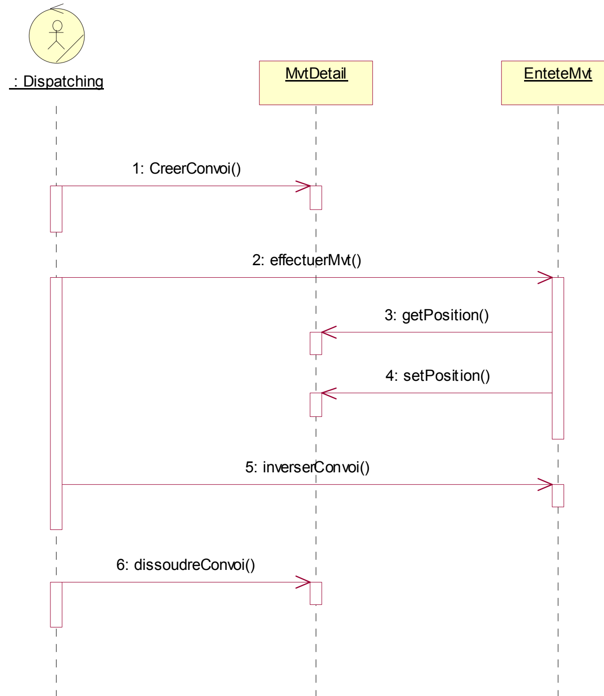

Figure 34 : Diagramme de séquence *Effectuer mouvement*

7.4 Méthodes de la bascule

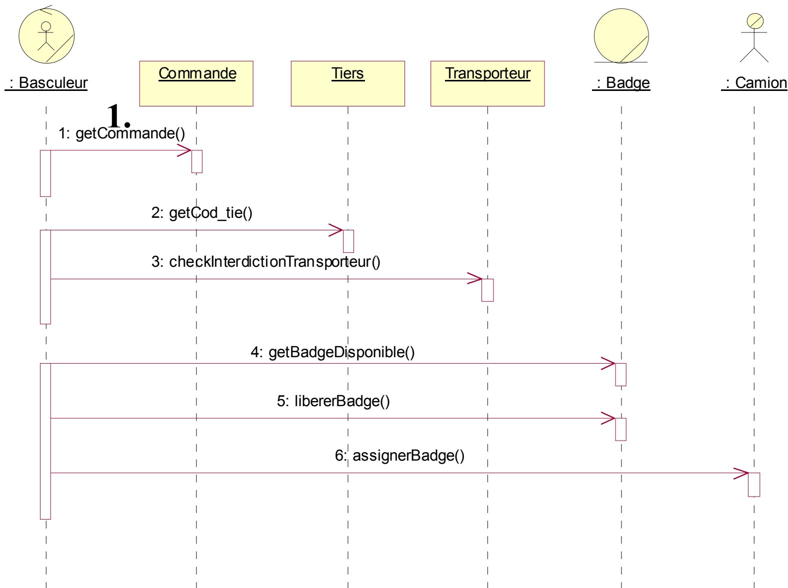

Figure 35 : Diagramme de séquence *Attribuer un badge*

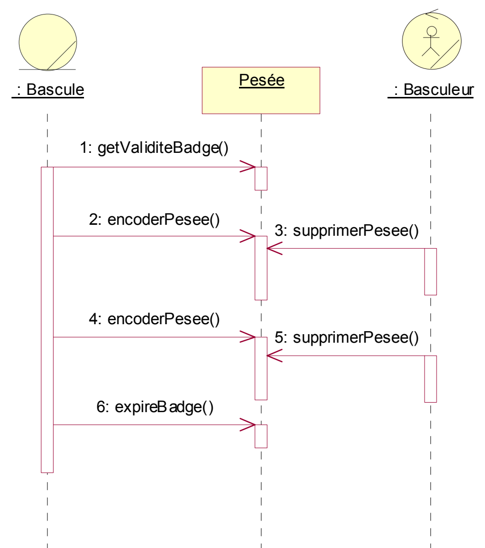

Figure 36 : Diagramme de séquence *Peser un véhicule*

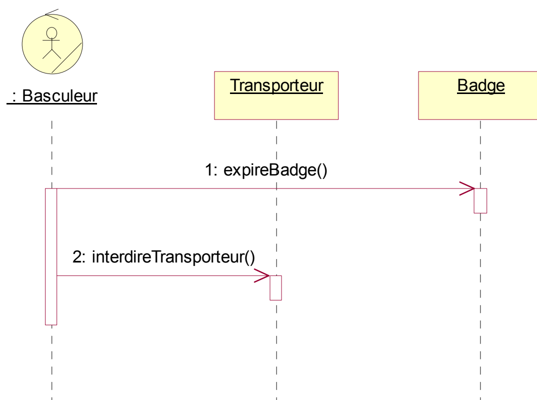

Figure 37 : Diagramme de séquence *Surveiller camion*

8 CONCLUSION

Ce document a présenté les activités de gestion des exigences de la cokerie, de modélisation métier de la traction et la bascule ainsi que l'analyse et la conception où ces différentes entités sont reliées. Ces activités de modélisation ont permis d'aborder une nouvelle dimension et de nouveaux domaines dans le projet Carsid. En effet, les travaux menés précédemment n'englobaient pas la dimension utilisateur du système informatique au niveau de l'entité cokerie et les flux de matières entrant et sortant de l'entreprise par train et par camion n'avaient pas été modélisés.

Ces divers aspects ont permis de dégager un ensemble de fonctions auxquelles le système informatique doit répondre. Ces fonctions peuvent être décomposées en sous-fonctions auxquels les méthodes de classe sont chargées de répondre. Ces méthodes ont ainsi pu être définies et spécifiées. Leur dynamique peut être visualisée dans les diagrammes de séquence. Chacun d'eux représente un ensemble de méthodes mises dans une certaine perspective pour répondre à une fonctionnalité.

Les travaux de modernisation du système informatique peuvent maintenant se concentrer sur d'autres aspects. Les activités de modélisation métier, de gestion des exigences, d'analyse et de conception ont été réalisées et il faut maintenant penser à l'implémentation du système. Aussi, le modèle est perfectible. Il est en effet possible d'approfondir les activités de modélisation notamment en travaillant plus amplement avec l'équipe informatique de Carsid pour obtenir une validation des modèles et de nouveaux détails a englober dans ceux-ci. Les travaux actuels n'ont également que peu abordé certains aspects tels que les co-produits et l'entretien où des travaux de modélisation pourraient être réalisés.

La prochaine étape du projet pourrait être de définir une interface graphique du futur système informatique reprenant toutes les fonctionnalités modélisées afin de pouvoir le soumettre aux utilisateurs finaux et bénéficier ainsi de leurs remarques et réflexions avant de passer à l'implémentation.

REFERENCES

G.Booch, I.Jacobson, James Rumbaugh, *The unified modelling language user guide*, Addison Wesley Pub Co, 1998.

A. Donnay, M. Kolp, D. Massart, T. T. Do, S. Faulkner, and A. Pirotte. *Modélisation orientée objet de la cokerie de CARSID*. Rapport final Carsid, Août 2002.

A. Donnay, F. Fouss, M. Kolp, D. Massart, A. Pirotte, Analyse orienté objet de processus sidérurgiques de type cokier, Working Paper IAG 86/03, Université Catholique de Louvain, Mars 2003.

F. Fouss, M. Ibarz, M. Kolp, *Object-oriented reengineering of the steel production databases at Carsid,* Working Paper IAG 85/03, Université Catholique de Louvain, Mars 2003.

P. Kruchten, *The Rational Unified Process An Introduction. Second Edition*, Addison-Wesley, Upper Saddle River, NJ, Juin 2001.

G. Lorge, *Informatisation de la traction* (document interne), CARSID, Marchinne-au-Pont, Octobre 2003.

L. Louvigny, *Modélisation orienté-objet d'aspects comportementaux de base de données. Application à la cokerie de Carsid*, Mémoire-projet IAG-UCL, Louvain-la-Neuve, 2003.

J. Rumbaugh, G. Booch, I. Jacobson, *The unified modelling language reference manual (Addison-Wesley object technology series)*, Addison Wesley Pub Co, 1998.

Y. Wautelet, *Application de la méthodologie RUP/UML à l'entreprise sidérurgique Carsid*, Mémoire-projet IAG-UCL, Louvain-la-Neuve, 2003.

Pays
cod_pay : Integ...
pay : String

Localite
cod_loc : Strin...
loc : String

Client
(from ConceptualModelPck)

Commande
num_com : Integer
ent_st : Boolean
getCommande()

Post
pst_num : Inte...
dat_pst : Date

Fam_article
fam_art : Strin...
lib_fam : Strin...
ori : String

Charbon
Number : Integer
Nom : String

tiers
cod_tie : Integer
nom_tie : String
adr_tie : String
ori : String
getCod_tie()

Fournisseur
(from ConceptualModelPck)

concerne
Test2 : Stri...

Pesee
num : Integer
DateTime : Date
pesee1 : Integer
pesee2 : Integer
pes_val : Boolean
pes_ann : Boolean
rem : String
rem_sys : String
status : String
lib_pre : String
cod_pre : Integer
encoderPesee()
supprimerPesee()

Bascule
bas_id : Integ...
bas : String

Livraison
cod_liv : Integ...
lib_liv : String
ori : String

Article
cod_art : Integer
cod_env : String
lib_art : String
libelle : String
ori : String

Transporteur
interdit : String
checkInterdictionTransport
interdireTransporteur()

Badge
num_bad : Integer
dat_cre : Date
dat_mod : Date
typ_bad : String
num_pas : Integer
pas_max : Integer
cod_ina : Boolean
ver_com : String
rec_bad : String
rem : String
rem_com : String
valide : Boolean
der_pes : String
bad_act : Boolean
expireBadge()
getBadgeDisponible()
libererBadge()

Livere
Test2 : String

Camion
pla_cam : String
chg_max : Integer
tare : Integer
frc_tar : Integer
dat_tar : Date
der_tar : Date
assignerBadge()

Envoi
Numero : Int...

Bateaux
Nom : String

Bassin
Bassin : Long
Libelle : String

Site
Site : Long
Libelle : String

Usine
Usine : Long

Service
Service : Long
Libelle : String

Raccordement
RaccSncb : String
Libelle : String
RaccCklSam : Boole...

Voie
Voie : String
IndexVoie : Long
Type : String
Libelle : String
Orientation : String
SensPriv : String
Etat : String
Longueur : Long
CulDeSac : Boole...

MvtDetail
DateTime : Date
Position : Integer
NumLVoit : Integer
PoidsNet : Long
PoidsBrut : Long
PoidsTheorique : Long
Tare : Long
Statut : String
creerConvoi()
dissoudreConvoi()
getPosition()
setPosition()

EnteteMvt
NumMvt : Long
TypeMvt : String
Intercaler : String
Inversion : String
TypeMvtSens : String
DateEnreg : Date
DateDebut : Date
DateFin : Date
VoieDespose : String
SensDepose : String
inverserConvoi()
effecturMvt()

Matiere
Matiere : String
Libelle : String

TypeMatiere
TypeMatiere : Strin...
Libelle : String
Couleur : String

Wagon
NumWagon : Long
PoidsBrut : Long
Tare : Long
PoidsNet : Long
PoidsTheorique : Long
Longueur : Long
DateTimeDebChomage : Date
NumChassis : Integer
setDebutChomage()
setFinChomage()
calculerTempsChomage()

EtatWagon
Etat : String
Libelle : String

TypeWagon
TypeWagon : Strin...
Libelle : String
Couleur : String

CategChomage
CategChomage : String
Libelle : String
tempsChomageMax()

GroupLoco
Numero : Long

Locomotive
NumLoco : Long

Tender
Tender : Long

de type 1
de type 2
de type
dans
conduit
accompagne
Origen

AnalyseCharbon
Quantite : Long

Element
Nom : String

AnalyseCoke
Quantite : Long

AnalyseMelange
Quantite : Long

Granulometrie
Interval : String
Nom : String

Melange
Numero : Integer
SwellingIndContMax : Long
SwellingIndDilMax : Long
Nom : String

GranCoke
Quantite : In...

GranMelange
Quantite : In...

MsgCommunication
Auteur : String
Dh_DebutAffichage : Da...
Dh_FinAffichage : Date
Commentaire : String
Response : String

MsgGroupe
Id_GroupMsg : Integer
LibelleMsg : String
TypeMsg : String

Sales
Poids : Long
DateTime : Date

MvtCharbon
Numero : Integer
Poids : Long
DateTime : Date

MvtMelange
Numero : Integer
Poids : Long
DateTime : Date

MvtCoke
Numero : Integer
Poids : Long
DateTime : Date
Retoum : Boolean

ComposeMelange
Quantite : Integer

SubMvtCharbon
Numero : Integ...
Poids : Long
DateTime : Date

SubMvtMelange
Numero : Integer
Poids : Long
DateTime : Date

SubMvtCoke
Numero : Integer
Poids : Long
DateTime : Date

Coke
Numero : Integer
RSID_I10 : Long
RSID_I20 : Long
RSID_I40 : Long
Nom : String

Entreprise
Nom : String

PlaceOfCharbon
(from ConceptualModelPck)

PlaceOfMelange
(from ConceptualModelPck)

PlaceProduction
Code : String

PlaceOfCoke
(from ConceptualModelPck)

Enfournement
Dh_Enf : Date
Dh_Enf_Divers : Date
enfournementE3()
enfournementE1()
txUtilisationE3()

Enfournement_Divers
V001 : Integer
V100 : Integer
Dh_Pause : Date
Dh_Jour : Date
Dh_Mois : Date

Defourneuse
Id_Defourneuse : Integer
LibDefourneuse : String
Id_Engin : Integer
PuissanceMaximale : Integer
PuissanceCalculPad : Integer
PuissanceCalculTGr : Integer
CoefCorrMul : Integer
CoefCorrAdd : Integer
NbrSecondeEchantillonnage : Inte...
TpsMaxDefournament : Integer
SurfaceRef : Integer

Defournement
Dh_Def : Date
Dh_Def_Cotherm : Date
Dh_Def_Puissance : Date
Dh_Def_Profil : Date
Dh_DefPuissanceCale : Date
Dh_Def_Planifee : Date

Defournement_Cotherm
Integer
Integer
enregistrementTemperatureCotherm()
CothermBatterie()
CothermFour()
MoyenneTempCotherm()
ecartTypeTempCotherm()
coubesTemperatureSaum...
historiqueCoubesTemperat...
checkCameaux()

Defourmament_Puissance
V001 : Integer
V113 : Integer
enregistrementPuissance()
cartePuissanceBatterie()
cartePuissanceFour()
evolution3DPic()
evolution3DPAD()
MoyenneBatterie()
...ournements()
...uissanceMois()

Defournement_ProfilCoke
V001 : Integer
V088 : Integer
enregistrementProfilCoke()
encoderCahierElectronique()

Figure 38 : Diagramme de classe – première partie

MsgCommunication
Auteur : String
Dh_DebutAffichage : Da...
Dh_FinAffichage : Date
Commentaire : String
Response : String

MessageRejete
Id_Msg : Integer
DateTraitementMsg : Da...
NumeroErreur : Integer
DescriptionErreur : String
EmetteurMsg : String
NumeroMsg : Integer
LibelleMsg : String
SourceErreur : String

MessageTraceChronologiqueData
DateTraitementMsg : Date
EmetteurMsg : Date
NumeroMsg : Date
NumeroSec : Date
EnteteMsg : String
Dh_Data : Date
Dh_Pause : Date
Dh_Jour : Date
Dh_Mois : Date

TCxListeSerieTempPyrometre
Id_Serie : Integer
TypeDeGaz : String
PCI : Integer
Dh : Date
Pyrometreur : String
Regleur : String
Commentaire : String
Valide : Boolean
encoderSerieTempCx()
validerSerie()
encoderCommentaire()
historiqueTCx5-22()

TCxSerieTempPyrometre
NumCx : Integer
Temperature : Integer
enregistrerTempCx()
getTempSerie()

TcxTypeDeMesure
Id_TypeDeMesure : Integ...
Libelle : String

Place
Name : Stri...

Deplacer
Dh_DepartGoud : Date
RetourEauCamion : Boolean
Type : String
FirmeEntreeFuel : String
HteurAvantEntreeFuel : Sing...
HteurApresEntreeFuel : Single
NoResEntreeFuel : Single
LitrageEntreeFuel : Single
HeureEntree : String

CamionCoProduit
NoPlaque : String

RecapPauseCoProduit
DhPause : Date
ExtracteurNord : Single
ExtracteurSud : Single
VitesseNord : Single
VitesseSud : Single
DebitNord : Single
DebitSud : Single
TemperatureVoluteNord : Single
TemperatureVoluteSud : Single
AmperageNord : Single
AmperageSud : Single
RegulationNord : String
RegulationSud : String
VanneRappoldNord : String
VanneRappoldSud : String
PressionHuileEntreeNord : Single
PressionHuileEntreeSud : Single
PressionHuileSortieNord : Single
PressionHuileSortieSud : Single
VibrationHorizontaleNord : Single
VibrationHorizontaleSud : Single
VibrationVerticaleNord : Single
VibrationVerticaleSud : Single
TemperatureExterieureNord : Single
TemperatureExterieureSud : Single
CondensationDebit : Single
CondensationAnalyse : Single
LavageDebit : Single
LavageAnalyse : Single
CSFDebit : Single
CSFAnalyse : Single
BiologiqueDebit : Single
BiologuiqueAnalyse : Single
VenturiDebit : Single
VenturiTemperature : Single
VenturiPh : Single
Brigadier : String
Operateur : String
OperateurGazometre : String
ItinerantCoproduit : String
ItinerantBioGazometre : String
ChaguerGourdon : String
HorsQuota : String
TempSortieCTH1 : Single
TempSortieCTH2 : Single
TempSortieCTH3 : Single
TempSortieCTH4 : Single
TempSortieEGR1 : Single
TempSortieEGR2 : Single
TempSortieEGR4 : Single
TempSortieEGR3 : Single
TempEntreeLaveur5Nord : Single
TempEntreeLaveur5Sud : Single
TempEntreeLaveur1Nord : Single
TempEntreeLaveur1Sud : Single
PressionEtDebitVapeur : String
NiveauRes500 : Single
NiveauResB500 : Single
NiveauRes1000 : Single
NiveauRes800 : Single
NiveauRes6514 : Single
PosteEmetteur : String
DetecteurCO : String
Commentaire : String
SignatureBrigadier : String
CommentaireOperateur : String
SignatureOperateurTechnique* : String
VolumeResFuel6509 : Single
HauteurResFuel6509 : Single
VolumeResFuel6508 : Single
HauteurResFuel6508 : Single
VolumeResFuel6511A : Single
HauteurResFuel6511A : Single
VolumeResFuel6511B : Single
HauteurResFuel6511B : Single
Laveur1NordDebit : Single
Laveur1NordPassage : Single
Laveur1SudDebit : Single
Laveur1SudPassage : Single
PerteFuel : Single
HauteurSoudeAvant : Single
HauteurSoudeApres : Single
ConsommationSoude : Single
encoderCahierElectronique()

MsgGroupe
Id_GroupMsg : Integer
LibelleMsg : String
TypeMsg : String

Four
Id_Four : Inte...

Piedroit
Id_Pdt : Integ...
Libelle : String

PdefDHPourDebug
DhMaintenant : D...
DhAvant : Date

ConsigneTempsCuisson
_Consigne : Integer
h_DebutChangement : Date
ConsTpsCuis_KoppersDidier : Integer
ConsTpsCuis_Coppee : Integer

Batterie
Id_Batterie : Integer
Libelle : String
Id_PdtDebut : Integer
Nom : String
NumFourDeb : Integer
NbrFour : Integer
NbrPdt : String

PdelListeHeurePourModeleLight
Id_Progr : Integer
HeureFour : Date

PlanningEntretien
Dh_Debut : Date
DureeEnMinute : Integer

RecapJourCoProduit
DhJour : Date
VolumeRes2Goudron : Integer
VolumeRes3Goudron : Integer
VolumeRes4Goudron : Integer
VolumeRes5Goudron : Integer
VolumeRes6Goudron : Integer
VolumeRes7Goudron : Integer
VolumeRes8Goudron : Integer
VolumeRes9Goudron : Integer
VolumeRes10Goudron : Integer
CpteurFuelVersCentralAvant : Integer
CpteurFuelVersCentralApres : Integer
LitrageFuelVersCentral : Integer

PDefParametre
Id : Integer
DateDemiereModif : Date
Commentaire : String
Dh_PauseEnCours : Date
TpsCuissonCoppee : Integer
TpsCuissonKoppers : Integer
DelaiEntreDefoumEtEnfoum : Integer
EcartMaxTpsCuissonCoppee : Integer
EcartMaxTpsCuissonKoppers : Integer
EcartSurProgrToleréCoppee : Integer
EcartSurProgrToleréKoppers : Integer
DefaultDelaiEntreDefEtEnf : Integer
ModifDhMachineAutorisee : Integer
DelaiEnSecondeDuRefreshAutoInterface : Integer
TimeOutExecuteModele : Integer
NbrHeureAvantPauseCouranteAVisualiser : Integer
NbrHeureApresPauseCouranteAVisualiser : Integer
PasswordOption : String
AnimationVideo : Integer
M_PV_Enabled : Integer
M_PV_kdpapHS : Integer
EntretienHeureMin : Integer
EntretienHeureMax : Integer
EntretienDureeMin : Integer
EntretienDureeMax : Integer
EntretienPasMin : Integer

RecapPauseFour
...ause : Date
...Ajusteur : String
...mentaireAjusteur : String
...atureAjusteur : String
...Electricien : String
...mentaireElectricien : String
...atureElectricien : String
...Contremaitre : String
...tantContremaitre : String
...mentaireContremaitre : String
...atureContremaitre : String
...DefoumementCoppee : Single
...DefoumamentKoppers : Single
...DefoumamentDidier : Single
...CuissonCoppee : String
...CuissonKoppers : String
...CuissonDidier : String
...eauCoppeeNettoyeCoteCoke : String
...eauKoppersNettoyeCoteCoke : String
...eauDidierNettoyeCoteCoke : String
...eauCoppeeNettoyeCoteMachine : String
...eauKoppersNettoyeCoteMachine : String
...eauDidierNettoyeCoteMachine : String
...ExtinctionEnSeconde : Integer
...EgouttageEnSeconde : Integer
...ExpeditionCK16EnMinute : Integer
PompeEnService : String
CokecarEnService : String

Dh_Mesure : Date
V001 : Integer
V077 : Integer

MesureSynchroneDiversMinute
Dh_Mesure : Date
V001 : Integer
V106 : Integer

MesureSynchroneDiversPause
Dh_Mesure : Date
V001 : Integer
V100 : Integer

MesureSynchroneDiversJour
Dh_Mesure : Date
V001 : Integer
V065 : Integer

Enfournement
Dh_Enf : Date
Dh_Enf_Divers : Date
enfournementE3()
enfournementE1()
txUtilisationE3()

Enfournement_Divers
V001 : Integer
V100 : Integer
Dh_Pause : Date
Dh_Jour : Date
Dh_Mois : Date

PdefResultat
Dh_Def_Prevue : Date
Etat_Prevu : Integer
Etat_HS_Prevu : Integer
Dh_Def_PrevueCorrigee : Date
Etat_Corrige : Integer
Etat_HS_Corrige : Integer
Entretien : String
Commentaire : String
Dh_Enf_Prec : Date
Dh_Def_Minimale : Date
Operation : String
enregistrePlanningPropose()
enregistrePlaningValide()

PdefListeResult
Id_Resultat : Integer
Dh_IdResultat : Date
Dh_Pause : Date
TypeDeModele : String
NomMachineClient : String
NomUser : String
StatusExecutionModele : Integer
Dh_FinExecuteModele : Date
LibelleStatus_FinExecuteModele : String
CommentaireResultat : String
startModelePV()
startModeleLight()
setResultat()
imprimePlanning()

PdefEtat
Etat : Integer
Libelle : String
Signification : Stri...

PdefNombreFourProduit
Dh_Pause : Date
NbreFourCopModelise : Integer
NbreFourKDModelise : Integer
NbreFourCopProgramme : Integer
NbreFourKDProgramme : Integer
NbreFourCopRealise : Integer
NbreFourKDRealise : Integer

PlanningDefournement
Dh_Def_Planifiee : Date
Dh_Def_Validee : Date
Dh_Enf_Validee : Date
Dh_Def_Mach : Date
Dh_Enf_Mach : Date
Dh_Enf_Prec : Date
Etat : Integer
Etat_Prec : Integer
Etat_HS : Integer
Etat_RP : Integer
Dh_Def_Corrigee : Date
Dh_Enf_Corrigee : Date
Justificatif : Integer
Commentaire : String
Entretien : String
Flag_Dh_Def_Mach_MajAPRESCCorrection : Integer
Flag_Dh_Enf_MachAPRESCCorrection : Integer
validationEnfournement()
validationDefournement()
setPlanning()

Participer
Test : Stri...

PlanningPoste
Id_Cycle : Integ...
Equipe : Integer

Personne
Id_Personne : Integer
Prenom : String
Nom : String
DateNaissance : Date

Figure 39 : Diagramme de classe – deuxième partie

Annexe 1 : Spécification des méthodes de la cokerie.

Classe PdefListeResult

```
// Le contremaître décide de lancer le modèle
// Patricio Villa  pour le calcul de la planification
// des enfournements et des défournements pendant // la pause suivante.
// PRE : la pause précédente est terminée.
// POST : une planification des enfournements et
//          défournements pour la pause suivante a été
//          calculée par le modèle Patricio Villa ou un
//          message d'erreur est généré s'il n'a pas été
//          à même de le faire.
// RETURN : un objet de type PdefListeResultat
//               contenant la planification si elle a pu
//               être calculée, null sinon.
PdefListeResult startModelePV()

// Le contremaître décide de lancer le modèle
// dégradé pour le calcul de la planification des
// enfournements et des défournements pendant la
// pause suivante.
// PRE : la pause précédente est terminée et le
//          modèle Patrico Villa n'a pas fourni de
//          résultat.
// POST : une planification par défaut (un
//          défournement toutes les sept minutes)
//          pour une pause a été calculé.
// RETURN : un objet de type PdefListeResultat
//               contenant la planification par défaut.
PdefListeResult startModeleLight()

// Le contremaître encode les causes qui ont
// provoqué les défournements qui se sont passés
// trop tôt ou trop tard par rapport au planning prévu
// lors de la pause précédente.
// PRE : fin de pause et un retard a eu lieu dans le
//          planning des défournement de la pause
//          précédente.
// POST : le champ CommentaireResultat pour
//          l'objet PdefListeResult correspondant à la
//          pause précédente à été mis à jour.
// PARAM : - Id_Resultat, l'identifiant unique de
//               l'objet PdefListeResultat dont le
//               commentaire doit être mis à jour.
//             - CommentaireResultat, une chaîne de
//               caractères contenant la cause du retard
//               des défournements.
void setResultat(Id_Resultat, CommentaireResultat)

// Une fois le planning défournement validé le
// contremaître procède à son impression.
// PRE : un planning défournement validé.
// POST : aucun.
// PARAM : un objet de la classe
//             PdefListeResult instancié à la
//             planification validée par le
//             contremaître.
void imprimePlanning(PdefListeResult)
```

Classe PdefResultat

```
// Enregistrement dans la base de données du
// planning proposé par le modèle Patricio Villa ou
// le modèle Light.
// PRE : un modèle a tourné et fournit un résultat.
// POST : le résultat du modèle a été enregistré dans
//          un objet instance de la classe
//          PdefResultat qui a été créé.
// PARAM : un objet de la classe
//             PdefListeResult instancié à la
//             planification fournie par le modèle.
void enregistrePlanningPropose(PdefListeResult)

// Le contremaître valide le modèle qu'on lui a
// proposé et qu'il a éventuellement modifié.
// PRE : un modèle a tourné et a renvoyé une
//          proposition de planification.
// POST : la planification validée est enregistrée
//          dans l'objet de type PdefResultat
//          contenant la planification proposée pour la
//          pause concernée.
// PARAM : un objet de la classe
//             PdefListeResult instancié à la
//             planification validée par le
//             contremaître.
void enregistrePlaningValide(PdefListeResult)
```

Classe Enfournement

```
// Lors d'un enfournement par l'enfourneuse E3, ses
// automatismes fournissent des données de
// production (heure, n° four, poids, humidité de la
// pâte).
// PRE : un enfournement a eu lieu sur le four
//          identifié.
// POST : un nouvel objet instance de la classe
//          Enfournement a été créé sur base des
//          données transmises.
// PARAM : id_four, l'identifiant du four sur lequel
//             a lieu l'enfournement.
void enfournementE3(id_four)

// Lors d'un enfournement par l'enfourneuse E1,
// l'ordinateur prend en compte des valeurs par
// défaut.  Il détecte qu'un enfournement E1 a eu lieu // lors de l'opération de repallage.
// PRE : un repallage a lieu sur un four identifié sur
//          lequel aucun enfournement n'a été
//          récemment enregistré.
// POST : un nouvel objet instance de la classe
//          Enfournement a été créé sur base de
//          données par défaut.
// PARAM : id_four, l'identifiant du four sur lequel
//             a lieu l'enfournement.
void enfournementE1(id_four)

// A partir des données fournies par l'enfourneuse E3
// et la repaleuse, le calculateur détecte les périodes
// d'utilisation des enfourneuses.
// PRE : les enfournements ont été encodé.
// POST : aucun.
// RETURN : un nombre réel représentant le taux
//               d'utilisation de l'enfourneuse E3.
float txUtilisationE3()
```

Classe PlanningDefournement

```
// Le contremaître valide, via l'intranet,
// l'enfournement qui a eu lieu.
// PRE : un enfournement a eu lieu et n'a pas encore
//          été validé.
// POST : cet enfournement est marqué valide dans
//          la base de données car l'attribut
//          Dh_Enf_Validee a été affecté à la date et à
//          l'heure de l'enfournement.
// PARAM : le champ de type Date Dh_Enf qui est
//             l'identifiant unique d'un enfournement.
void validationEnfournement(Dh_Enf)
```

```
// Le contremaître valide, via l'intranet, le
// défournement qui a eu lieu.
// PRE : un défournement a eu lieu et n'a pas encore
//        été validé.
// POST : ce défournement est marqué valide dans la
//         base de données car l'attribut
//         Dh_Def_Validee a été affecté à la date et à
//         l'heure du défournement.
// PARAM : le champ de type Date Dh_Def qui est
//          l'identifiant unique d'un défournement.
void validationDefournement(Dh_Def)

// A la fin du traitement de la méthode
// enregistrePlanningValide() de la classe
// PdefResultat, cette dernière appelle la méthode
// setPlanning() pour enregistrer le planning de la
// pause suivante dans un objet de type
// PlanningDefournement.
// PRE : un planning a été validé et enregistré dans
//        PdefResultat.
// POST : un objet PlanningDefournement est créé et
//         le planning de la pause suivante y est
//         enregistré.
// PARAM : Liste_Dh_Enf_Def[] une liste chaînée
//          des enfournements et défournements qui
//          viennent d'être planifiés.
void setPlanning(Liste_Dh_Enf_Def[])
```

Classe TCxListeSerieTempPyrometre

```
// L'équipe réglage mesure la température dans
// certains carneaux durant la cuisson et les encode
// via l'intranet.
// PRE : une nouvelle série de températures doit être
//        encodée.
// POST : un nouvel objet
//         TcxListeSerieTempPyrometre dont
//         l'instance comprend les données sur la
//         série de températures relevées a été créé
//         (ce ne sont pas les températures elles-
//         mêmes au niveau de cette classe).  Les
//         températures relevées sont également
//         enregistrées dans la base de données.
// PARAM : - TypeDeGaz, PCI, Dh, Pyrometreur,
//            Regleur des informations sur la série
//            de températures relevées.
//          - TableauPiedroitCxTemp[][][] un
//            tableau dont les trois dimensions
//            contiennent respectivement
//            l'identifiant du piédroit, le numéro du
//            carneau sur le piédroit et la
//            température relevée.
void encoderSerieTempCx(TypeDeGaz, PCI, Dh, Pyrometreur,
Regleur, TableauPiedCxTmp)

// L'équipe réglage peut valider une série (une série
// est une série de température des carneaux relevée)
// après son enregistrement.
// PRE : une série est enregistrée et non validée
//        (l'attribut valide vaut false).
// POST : l'attribut valide vaut true.
// PARAM : id_Serie, l'identifiant de la série à
//          valider.
void validerSerie(id_Serie)

// L'équipe réglage peut encoder un commentaire sur
// une série après son enregistrement.
// PRE : une série est enregistrée.
// POST : l'attribut Commentaire de l'objet
//         correspondant à cette série est mis à jour.
// PARAM : id_Serie, l'identifiant de la série à
//          valider
void encoderCommentaire(id_Serie)

// Cette méthode affiche un graphe sur base des
// températures des cinq dernières séries
// encodées dans la base de données.
// PRE : au moins cinq séries enregistrées dans la
//        base de données.
// POST : aucun.
void historiqueTCx5-22()
```

Classe TCxSerieTempPyrometre

```
// Cette méthode a pour effet d'enregistrer une liste
// de températures dans la base de données.
// PRE : l'existence d'un objet
//        TcxListeSerieTempPyrometre auquel
//        l'enregistrement de la température se
//        rapporte.
// POST : des objets de la classe
//         TCxSerieTempPyrometre pour chaque
//         température de la liste des températures
//         ont été créés.
// PARAM : - id_serie, l'identifiant de la série à
//            laquelle les températures se rapportent.
//          - TableauPiedroitCxTemp[][][] un
//            tableau dont les trois dimensions
//            contiennent respectivement
//            l'identifiant du piédroit, le numéro du
//            carneau sur le piédroit et la
//            température relevée.
void enregistrerTempCx(id_serie, TableauPiedroitCxTemp)

// Cette méthode renvoie le tableau de températures
// enregistrées pour la série id_serie.
// PRE : id_serie, une série de températures existante
//        dans la base de données.
// POST : aucun.
// PARAM : id_serie, l'identifiant de la série dont on
//          demande les températures.
// RETURN : TableauPiedroitCxTemp[][][] un
//           tableau dont les trois dimensions
//           contiennent respectivement
//           l'identifiant du piédroit, le numéro du
//           carneaux sur le piédroit et la
//           température relevée.
TableauPiedroitCxTemp getTempSerie(id_serie)
```

Classe Defournement_Cotherm

```
// Mesure automatique par le guide-coke de la
// température atteinte par le saumon de coke
// pendant le défournement à trois niveaux différents
// d'un four donné (système Cotherm).
// PRE : un défournement a eu lieu et les
//        informations relatives à la température à la
//        sortie de celui-ci n'ont pas été encodées.
// POST : les informations relatives à la température
//         ont été enregistrées dans la base de
//         données.
// PARAM : Dh_Def, la date et l'heure du
//          défournement sur lequel porte
//          l'enregistrement de la température qui
//          est son identifiant unique.
void enregistrementTemperatureCotherm(Dh_Def)

// Cette méthode affiche une carte Cotherm pour une
```

```
// batterie de fours sur base des données présentes
// dans la base de données et répondant aux
// paramètres encodés (méthode appelée lorsque
// l'option Four n'est pas cliquée dans l'interface
// graphique).
// PRE : les températures ont été encodées dans la
//        base de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//             dont l'utilisateur veut la carte Cotherm.
//           - Niveau, le niveau auquel on se situe.  //
Trois valeurs sont possibles : Inf, Med
//             et Sup.
//           - NbrCycle, le nombre de cycles (vaut 1
//             à un nombre maximum variable).
//           - DH_Preparation, la date et l'heure de
//             préparation.
void carteCothermBatterie(Id_Batterie, Niveau, NbrCycle,
DH_Preparation)

// Cette méthode affiche une carte Cotherm pour un
// four sur base des données présentes dans la base
// de données et répondant aux paramètres encodés
// (méthode appelée lorsque l'option Four est cliquée
// dans l'interface graphique).
// PRE : les températures ont été encodées dans la
//        base de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//             qui possède le four dont l'utilisateur
//             demande la carte Cotherm.
//           - Id_Four, l'identifiant du four dont
//             l'utilisateur veut la carte Cotherm.
//           - Niveau, le niveau auquel on se situe.
//             Trois valeurs possibles : Inf, Med et
//             Sup.
//           - NbrCycle, le nombre de cycles (vaut
//             1 à un nombre maximum variable).
//           - DH_Preparation, la date et l'heure de
//             préparation.
//           - DH_Defournement, la date et l'heure
//             du défournement pour lequel on
//             désire la carte Cotherm.
//           - Id_Graphique vaut 2 si l'utilisateur
//             désire un graphique sur 3 niveaux de
//             détails et vaut 3 si l'utilisateur veut
//             un graphique comprenant l'historique
//             des 5 derniers défournements.
void carteCothermFour(Id_Batterie, Id_Four, Niveau, NbrCycle,
DH_Preparation, Dh_Defournement, Id_Graphique)

// Cette méthode calcule la température Cotherm
// moyenne d'une batterie pour une préparation
// donnée.
// PRE : les températures ont été encodées dans la
//        base de données lors des défournements.
// POST : aucun
// PARAM : - Id_Batterie, l'identifiant de la batterie
//             pour laquelle on recherche la
//             moyenne.
//           - Dh_Preparation la date et l'heure de
//             préparation.
// RETURN : la température moyenne Cotherm pour
//             cette batterie.
float moyenneTempCotherm(Id_Batterie, Dh_Preparation)

// Cette méthode calcule l'écart type température
// Cotherm d'une batterie pour une préparation
// donnée.
// PRE : les températures ont été encodées dans la
//        base de données lors des défournements.
// POST : aucun
// PARAM : - Id_Batterie, l'identifiant de la batterie
//             pour laquelle on recherche l'écart type.
//           - Dh_Preparation la date et l'heure de
//             préparation.
// RETURN : l'écart type de la température Cotherm
//             pour cette batterie.
float ecartTypeTempCotherm(Id_Batterie, D Dh_Preparation)

// Cette méthode affiche le graphique de type
// courbes de la température du saumon de coke qui
// est basée sur un défournement si elle est appelée
// par le bouton four dans l'interface graphique et sur
// la préparation si elle est appelée par le bouton
// barre de l'interface graphique.
// PRE : les températures ont été encodées dans la
//        base de données lors des défournements.
// POST : aucun.
// PARAM : - Id_four, l'identifiant du four dont on
//             recherche les courbes de température
//             du saumon de coke.
//           - DH, la date et l'heure qui correspond
//             à Dh_Defonrnement, la date et
//             l'heure du défournement (qui est sont
//             identifiant unique) pour lequel on
//             recherche la courbe de température
//             du saumon de coke et correspond ou
//             à Dh_Preparation la date et l'heure de
//             préparation (en fonction de la valeur
//             de l'attribut ClickFour).
//           - ClickFour, un booléen valant true si la
//             méthode est appelée par le bouton
//             four et donc que DH est affecté à
//           - Dh_Defournement et valant false
//             sinon (dans ce cas DH est affecté à
//             Dh_Preparation).
void courbesTemperatureSaumon(Id_Four, Dh, ClickFour)

// Cette méthode affiche le graphique de type
// courbes de l'historique des températures du
// saumon de coke. Il en donne les x dernières
// valeurs.  Ce graphique est basé sur un
// défournement si la méthode est appelée par le
// bouton four dans le client graphique et un
// graphique basé sur la préparation si elle est
// appelée par le bouton barre du client graphique.
// PRE : les températures ont été encodées dans la
//        base de données lors des défournements.
// POST : aucun.
// PARAM : - Id_four, l'identifiant du four dont on
//             recherche les courbes de température
//             du saumon de coke.
//           - DH, la date et l'heure qui
//             correspondent à Dh_Defonrnement, la
//             date et l'heure du défournement (qui
//             est sont identifiant unique) pour
//             lequel on recherche la courbe de
//             température du saumon de coke et
//             correspond ou à Dh_Preparation la
//             date et l'heure de préparation.
//           - Niveau, le niveau auquel on se situe.
//             Trois valeurs possibles : Inf, Med et
//             Sup.
//           - ClickFour, un booléen valant true si la
//             méthode est appelé par le bouton four
```

```
//                 et donc que DH est affecté à
//                 Dh_Defournement et valant false
//                 sinon (dans ce cas DH est affecté à
//                 Dh_Preparation).
void historiqueCourbesTemperatureSaumon(Id_Four, DH,
Niveau, ClickFour)

// En utilisant la moyenne des températures pour
// chaque carneau enregistrées au cours de plusieurs
// défournements successifs, cette méthode vérifie si
// les carneaux présentent une anomalie durable (sur
// base du système Cotherm).
// PRE : les températures ont été encodées dans la
//         base de données lors des défournements.
// POST : aucun.
// PARAM : Id_Four, l'identifiant du four dont on
//             teste la normalité des carneaux.
// RETURN : true si les carneaux ne présentent pas
//              d'anomalie, false sinon.
boolean checkCarneaux(Id_Four)
```

Classe Defournement_Puissance

```
// Cette méthode procède à l'enregistrement de la
// puissance au défournement sur base des données
// fournies par la défourneuse.
// PRE : un défournement a eu lieu et les
//         informations relatives à la puissance de
//         celui-ci n'ont pas été encodées.
// POST : les informations relatives à la puissance
//          ont été enregistrées dans la base de
//          données.
// PARAM : Dh_Def, l'identifiant unique du
//             défournement sur lequel porte
//             l'enregistrement de la puissance.
void enregistrementPuissance(Dh_Def)

// Cette méthode affiche le graphique de la moyenne
// de la puissance au défournement pour une batterie
// de fours sur plusieurs défournements sur base des
// données présentes dans la base de données et
// répondant aux paramètres encodés (méthode
// appelée lorsque l'option Four n'est pas cliquée
// dans l'interface graphique).
// PRE : les puissances ont été encodées dans la base
//         de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//               pour laquelle on recherche la carte de
//               puissance au défournement.
//             - NbrCycle, le nombre de cycles (vaut 1
//               à un nombre maximum variable).
//             - Dh_Preparation la date et l'heure de
//               préparation.
void cartePuissanceBatterie(Id_Batterie, NbrCycle,
Dh_Preparation)

// Cette méthode affiche le graphique de la moyenne
// de la puissance au défournement pour un four sur
// plusieurs défournements sur base des données
// présentes dans la base de données et répondant aux
// paramètres encodés (méthode appelée lorsque
// l'option Four est cliquée dans l'interface
// graphique).
// PRE : les puissances ont été encodées dans la base
//         de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//              pour laquelle on recherche la carte de
//               puissance au défournement.
//             - Id_Four, l'identifiant du four dont
//               l'utilisateur veut la carte de puissance.
//             - NbrCycle, le nombre de cycles (vaut 1
//               à un nombre maximum variable).
//             - DH_Preparation, la date et l'heure de
//               préparation.
//             - DH_Defournement, la date et l'heure
//               du défournement pour lequel on
//               désire la carte Cotherm.
void cartePuissanceFour(Id_Batterie, Id_Four, NbrCycle,
DH_Preparation, Dh_Defournement)

// Cette méthode affiche un graphe sur l'évolution de
// l'intensité maximale (A) de défournement pour
// chaque four d'une batterie.
// PRE : les puissances ont été encodées dans la base
//         de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//               pour laquelle on recherche le graphe.
//             - DH_Preparation, la date et l'heure de
//               préparation.
void evolution3DPic(Id_Batterie, DH_Preparation)

// Cette méthode affiche un graphe de l'évolution du
// PAD pour chaque four d'une batterie. Le PAD est
// une sorte de mesure de la dispersion de l'intensité,
// alors que l'intensité maximale se rapproche d'une
// moyenne. Le PAD est défini comme le temps
// écoulé entre la première fois où l'intensité (A) lors
// du défournement atteint 30A (à l'ascendant), avant
// d'atteindre l'intensité maximale, et le moment où
// l'intensité revient à 30A (au descendant).
// PRE : les puissances ont été encodées dans la base
//         de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//               pour laquelle on recherche le graphe.
//             - DH_Preparation, la date et l'heure de
//               préparation.
void evolution3DPAD(Id_Batterie, DH_Preparation)

// Cette méthode affiche un graphe sur l'évolution de
// « l'aire reconstituée » pour chaque four d'une
// batterie.
// PRE : les puissances ont été encodées dans la base
//         de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//               pour laquelle on recherche le graphe.
//             - DH_Preparation, la date et l'heure de
//               préparation.
void evolution3DID(Id_Batterie, DH_Preparation)

// Cette méthode affiche un graphe sur l'évolution de
// la puissance moyenne au défournement sur une
// batterie.
// PRE : les puissances ont été encodées dans la base
//         de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
//               pour laquelle on recherche le graphe.
//             - DH_Preparation, la date et l'heure de
//               préparation.
void evolutionPuissanceMoyenneBatterie(Id_Batterie,
DH_Preparation)
```

// Cette méthode affiche un graphe sur les pics de
// puissance au défournement sur une batterie.
// PRE : les puissances ont été encodées dans la base
// de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
// pour laquelle on recherche le graphe.
// - DH_Preparation, la date et l'heure de
// préparation.
void picMoyDerniersDefournements(Id_Batterie, DH_Preparation)

// Cette méthode affiche un graphe sur le nombre de
// fours calés par mois et la puissance mensuelle.
// PRE : les puissances ont été encodées dans la base
// de données lors des défournements.
// POST : aucun.
// PARAM : - Id_Batterie, l'identifiant de la batterie
// pour laquelle on recherche le graphe.
// - DH_Preparation, la date et l'heure de
// préparation.
void nbrFoursCaleMoisPuissanceMois(Id_Batterie, DH_Preparation)

Classe Defournement_ProfilCoke
// Mesure automatique par le guide-coke du profil
// coke lors du défournement (système Cotherm).
// PRE : un défournement a eu lieu et les
// informations relatives au profil coke à la
// sortie de celui-ci n'ont pas été encodées.
// POST : les informations relatives au profil coke
// ont été enregistrées dans la base de
// données.
// PARAM : Dh_Def, la date et l'heure du
// défournement sur lequel porte
// l'enregistrement du profil coke et qui
// est son identifiant unique.
void enregistrementProfilCoke(Dh_Def)

Classe RecapPauseCoProduit
// Cette méthode permet à l'équipe contrôle en fin de
// pause d'encoder les données relevées au cours de
// la pause précédente.
// PRE : fin d'une pause.
// POST : une nouvelle entrée dans la table
// RecapPauseCoProduit contenant les
// informations encodées par l'équipe
// contrôle a été créée.
// PARAM : DhPause, la date et l'heure de la pause
// sur laquelle portent les données
// encodées, qui représente son identifiant
// unique.
void encoderCahierElectronique(DhPause)

Annexe 2 : Spécification des méthodes de la traction.

Classe Wagon
// Cette méthode a pour effet d'enregistrer la date et
// l'heure à laquelle arrive un wagon sur le site et
// d'enregistrer ainsi le début du temps de chômage.
// PRE : un wagon enregistré dans la base de
// données et qui n'est pas en chômage.
// POST : ce wagon est mis en chômage : son attribut
// DateTimeDebChomage est affecté au jour
// et à l'heure d'appel de la méthode.
// PARAM : NumWagon, l'identifiant unique du
// wagon qui doit être mis en chômage.
void setDebutChomage(NumWagon)

// Cette méthode permet de mettre le wagon hors
// chômage lorsqu'il est renvoyé à la SNCB.
// PRE : un wagon en chômage.
// POST : le wagon est mis hors chômage : son
// attribut DateTimeDebChomage est mis à
// zéro.
// PARAM : NumWagon, l'identifiant unique du
// wagon qui doit être mis hors chômage.
// RETURN : une date spécifiant le nombre de jours,
// d'heures et de minutes de chômage.
date setFinChomage(NumWagon)

// Cette méthode permet de calculer le nombre de
// jour de chômage afin de faire les calculs liés au
// paiement des amendes éventuelles.
// PRE : un wagon en chômage.
// POST : le résultat du calcul.
// PARAM : NumWagon, l'identifiant unique du
// wagon dont on recherche le temps de
// chômage.
// RETURN : une date spécifiant le nombre de jours,
// d'heures et de minutes de chômage.
date calculerTempsChomage(NumWagon)

Classe CategChomage
// Cette méthode permet de connaître le temps
// maximal de chômage pour un wagon donné.
// PRE : le wagon enregistré dans la base de données
// qu'il soit en chômage ou non.
// POST : aucun.
// PARAM : NumWagon, l'identifiant unique du
// wagon dont on recherche le temps de
// chômage maximum.
// RETURN : une date spécifiant le nombre de jours,
// d'heures et de minutes maximal de
// chômage.
date tempsChomageMax(NumWagon)

Classe MvtDetail
// Cette méthode permet de créer un convoi dans la
// base de données à partir d'une liste de wagons.
// PRE : un convoi physique a été créé à partir d'une
// liste de wagons existant dans la base de
// données.
// POST : le convoi a été enregistré dans la base de
// données : autant d'objets MvtDetail qu'il
// existe de wagons dans le convoi ont été
// créés.
// PARAM : - ListeWagons[], la liste des wagons qui
// constituent le convoi.
// - Position, un entier représentant la
// position du convoi créé sur la voie.
// Le champ position de chaque objet
// MvtDetail est affecté à la position
// donnée du convoi.
void creerConvoi(ListeWagons[], position)

// Cette méthode permet de détruire un convoi dans
// la base de données lorsqu'une dissolution
// physique d'un convoi a eu lieu.
// PRE : un convoi physique a été dissolu à partir
// d'une liste de wagons existant dans la base
// de données.
// POST : le convoi a été détruit de la base de
// données : les objets MvtDetail représentant
// chaque wagon du convoi ont été détruits.
// PARAM : ListeWagons[], la liste des wagons qui

```
//                constituaient le convoi.
void dissoudreConvoi(ListeWagons[])

// Cette méthode permet de connaître la position
// actuelle d'un convoi de wagons.
// PRE : un convoi de wagons ListeWagons[] dont
//          chaque wagon fait partie.
// POST : aucun
// PARAM : ListeWagons[] une liste chaînée de
//                NumWagons (identifiant unique d'un
//                objet wagon) représentant un convoi
//                physique de wagons.
// RETURN : position, un entier représentant la
//                position physique du convoi.
position getPosition(ListeWagons[])

// Cette méthode permet d'enregistrer une nouvelle
// position du convoi de wagons après un
// déplacement.
// PRE : un train a effectué un mouvement.
// POST : le mouvement a été encodé.
// PARAM : - ListeWagons[] l'ensemble des wagons
//                  formant le train qui doit être déplacé.
//               - Position, un entier représentant
//                  position d'arrivée du train.
void setPosition(ListeWagons[], Position)
```

Classe MvtDetail

```
// L'ordre des wagons apparaît sur le synoptique soit
// de droite à gauche (privilégié) soit de gauche à
// droite (non-privilégié). Lorsqu'un mouvement est
// effectué, les wagons se retrouvent parfois sur le
// synoptique dans un sens contraire à la réalité. La
// locomotive par contre se trouve toujours du côté
// gauche, quelle que soit sa position réelle.
// PRE : un convoi de wagons valide à inverser.
// POST : l'inversion a été enregistrée dans la base
//           de données.
// PARAM : ListeWagons[] l'ensemble des wagons
//                formant le train qui doit être inversé.
void inverserConvoi(ListeWagons[])

// Cette méthode permet d'enregistrer un mouvement
// après son encodage sur le synoptique.
// PRE : un train a effectué un mouvement.
// POST : le mouvement a été encodé.
// PARAM : - ListeWagons[] l'ensemble des wagons
//                  formant le train qui doit être déplacé.
//               - VoieArrive, la voie d'arrivée du
//                  convoi de wagons, la voie de départ
//                  étant la position du convoi au
//                  moment de l'appel de la méthode.
void effectuerMvt(ListeWagons[], VoieArrivee)
```

Annexe 3 : Spécification des méthodes de la bascule.

Classe Pesée

```
// Cette méthode permet d'encoder une pesée
// effectuée par la bascule dans la base de données.
// Le système détermine s'il s'agit de la première ou
// deuxième pesée de la paire de pesées.
// PRE : une pesée a été effectuée et doit être
//          enregistrée dans la base de données.
// POST : la masse du camion à vide ou à plein est
//           enregistrée.
// PARAM : - num, l'identifiant unique de la paire
//                  de pesées.
//               - masse, un entier représentant la masse
//                  en kilos de la pesée effectuée.
void encoderPesee(num, masse)

// Cette méthode supprime la dernière pesée encodée
// sur la paire de pesées d'identifiant num.
// PRE : au minimum une pesée a été encodée.
// POST : la dernière pesée encodée a été supprimée
//           de la base de données.
// PARAM : num, l'identifiant unique de la paire de
//                pesées.
void supprimerPesee(num)

// Cette méthode permet de passer un badge dans
// l'état expiré dans la base de données après un
// nombre déterminé de pesées.
// PRE : un badge a effectué le nombre de pesées
//          maximum durant lequel il est valide.
// POST : l'attribut valide pour ce badge est mis à
//           false.
// PARAM : num_bad, le numéro du badge qui est
//                son identifiant unique.
void expireBadge(num_bad)

// Cette méthode retourne un objet de type badge
// correspondant à un badge physique que le
// basculeur peut assigner à un camion.
// PRE : un camion sans badge valide.
// POST : aucun.
Badge getBadgeDisponible()

// Cette méthode permet de valider un badge qui ne
// l'était plus dans la base de données.
// PRE: un badge invalide (son attribut Valide vaut
//         false).
// POST : le badge est valide dans la base de données
//           (son attribut Valide vaut true).
// PARAM : num_bad, le numéro du badge qui est
//                son identifiant unique.
void libererBadge(num_bad)

// Cette méthode permet de vérifier la validité d'un
// badge.
// PRE : un badge d'identifiant num_bad existant
//          dans la base de données.
// POST : aucun.
// PARAM : num_bad, le numéro du badge qui est
//                son identifiant unique.
// RETURN : un booléen valant true si le badge est
//                valide et false sinon.
boolean getValiditeBadge(num_bad)
```

Classe Camion

```
// Cette méthode permet d'assigner un badge à un
// camionneur pour un certain nombre de pesées
// déterminé par le basculeur.
// PRE : un camion sans badge valide.
// POST : un camion avec un badge valide.
// PARAM : - num_bad, le numéro du badge qui est
//                  son identifiant unique.
//               - nbrPesees, le nombre de paire de
//                  pesées durant lequel le badge est
//                  valide.
void assignerBadge(num_bad, nbrPesees)
```

Classe Tiers

```
// Cette méthode permet d'obtenir des informations
// sur une partie tierce, c'est-à-dire, un transporteur,
// un client, un fournisseur.
```

```
// PRE : un camionneur sans badge.
// POST : aucun.
// PARAM : nom_tie, le nom de la partie tierce
//              (transporteur, client, fournisseur).
// RETURN : un objet de type Tiers contenant les
//               informations recherchées sur la partie
//               tierce.
Tiers getCod_tie(nom_tie)
```

Classe Transporteur

```
// Cette méthode permet de tester si un transporteur
// est interdit de traiter avec l'entreprise.
// PRE : un transporteur présent dans la base de
//          données.
// POST : aucun.
// PARAM : cod_tie, l'identifiant unique d'un tiers
//              dans la base de données.
// RETURN : un booléen valant true si le
//               transporteur n'est pas admis à traiter
//               avec l'entreprise, false sinon.
boolean checkInterdictionTransporteur(cod_tie)

// Cette méthode permet de marquer un transporteur
// interdit dans la base de données.
// PRE : un transporteur enregistré dans la base de
//          données.
// POST : le transporteur est marqué interdit dans la
//           base de données.
// PARAM : cod_tie, l'identifiant unique d'un tiers
//              dans la base de données.
void interdireTransporteur(cod_tie)
```

Classe Commande

```
// Cette méthode renvoie un objet de type
// Commande contenant les spécificités d'une
// commande au départ d'un numéro de commande.
// PRE : une commande d'identifiant num_com
//          existant dans la base de données.
// POST : aucun.
// PARAM : num_com, l'identifiant unique d'une
//              commande dans la base de données.
// RETURN : un objet de la classe Commande
//               contenant les informations sur la
//               commande.
Commande getCommande(num_com)
```

Annexe 4 : Critique d'UML.

UML est indiscutablement l'outil de modélisation qui a gagné aujourd'hui un plus large public. Néanmoins, cet outil n'est pas parfait, et de judicieuses critiques ont déjà permis et devraient encore permettre une amélioration du langage de modélisation.
Il n'y a rien de surprenant qu'UML, créé en 1996, soit toujours dans un processus de perfectionnement. De plus, rappelons qu'UML a succédé à plusieurs méthodes différentes, et qu'il est donc un compromis entre les vues de ses trois auteurs : Booch, Rumbaugh et Jacobson
Tout l'intérêt d'une critique d'un langage de modélisation est de permettre aux auteurs de perfectionner leur outil, et d'attirer l'attention de l'utilisateur sur les difficultés qu'il pourrait rencontrer.
Les vues qui vont être présentées ne se veulent pas exhaustives. Elles présentent néanmoins des pistes de réflexion sur le sujet, et devraient permettre une meilleure compréhension des caractéristiques d'UML.

Sémentique UML. Certains scientifiques, tels Andy Evans et Stuart Kent, proposent une approche plus précise de la sémantique d'UML, qui devrait apporter de nouveaux avantages au langage.
En effet, l'« Object Management Group » a contribué à la définition standard d'UML par un document sur sa sémantique. Celui-ci est constitué de trois parties : une syntaxe, des règles de formation correcte et une sémantique des éléments de modélisation. La syntaxe est exprimée sous formes de notations utilisant UML. Les règles de formation correcte sont quant à elles exprimées en utilisant OCL (« Object Constraint Language »). Finalement, la sémantique des éléments de modélisation est simplement décrite en langage naturel, c'est-à-dire en anglais.
D'après eux, cette description est trop informelle et pas assez structurée pour développer une analyse formelle d'UML et l'utilisation de techniques de développement. Rumbaugh lui-même dénonce cette lacune. Une sémantique précise d'UML aurait de nombreux avantages. Elle serait par exemple un point de référence en cas de confusion concernant le sens précis d'une construction. Elle permettrait aussi une base de comparaison univoque entre UML et d'autres techniques et notations. Elle apporterait aussi plus de cohérence entre les divers constituants d'UML, présents et à venir.
Un autre bénéfice majeur de cette approche serait la possibilité d'une analyse rigoureuse de certaines propriétés d'un modèle décrit en utilisant UML, et la réalisation de preuves justifiant un choix plutôt qu'un autre.
Enfin, de nouveaux outils pourraient être développés qui utiliseraient cette sémantique rigoureuse, comme par exemple des générateurs de code ou un vérificateur de cohérence.
C'est donc pour atteindre plus de précision dans la sémantique d'UML que le groupe pUML (« precise UML ») a été créé en 1997. C'est un groupe international de chercheurs et de praticiens ayant pour but de faire d'UML un langage de modélisation formel. Notons qu'ils ont choisi de poursuivre dans la lignée de la sémantique actuelle d'UML plutôt que de développer un modèle sémantique complètement neuf.
En cinq années d'existence, le groupe pUML a organisé toute une série de conférences et d'ateliers ayant pour sujet le développement d'UML en tant que langage de modélisation précis. Bien que de nombreux progrès aient déjà été réalisés, des avancées restent à faire. Récemment, plusieurs membres de pUML travaillent sur le MDA (« Model-Driven Architecture ») car celui-ci semble apporter des solutions intéressantes pour plus de précision dans les définitions.

Raffinement du modèle. Les notations d'UML permettent de représenter de façon très intuitive les objets et leurs interactions dans le but de développer par la suite un système qui aura les fonctionnalités souhaitées.
Une des forces d'UML est de permettre l'élaboration de modèles à différents niveaux. On peut prendre pour exemple les deux diagrammes de cas d'utilisation de la cokerie de Carsid. Le premier diagramme, généré durant la première phase du projet, est une modélisation métier qui a pour but de donner une vue d'ensemble du processus industriel concerné. Par contre le diagramme généré lors de la phase de gestion des exigences se veut beaucoup plus concret, et proche des besoins des utilisateurs.
Cependant, les connections à un niveau plus global des différentes représentations d'un même système posent quelques problèmes. Comme le notent Martin Hitz et Gerti Kappel, il serait essentiel de pouvoir vérifier la cohérence entre plusieurs modèles représentant des caractéristiques différentes mais complémentaires d'un même système.
Le problème devient encore plus délicat lorsque l'on réalise que bien souvent les besoins des utilisateurs changent au cours de la période de modélisation. Dès lors des mécanismes évolutifs

devraient être trouvés afin d'apporter plus de cohérence dans l'élaboration des modèles.
Le raffinement des éléments d'un modèle, qu'ils soient à différents niveaux de représentation, ou à un même niveau mais à différents moments dans le temps, se révèle important pour augmenter l'efficacité de la modélisation avec UML. Un tel raffinement doit augmenter la traçabilité des liens entre différents éléments d'un modèle, grâce par exemple à des stéréotypes qui permettraient de relier graphiquement les éléments d'un modèle ayant un lien entre eux.
C'est déjà en quelque sorte ce que les stéréotypes « include » et « extend » remplissent au niveau des diagrammes de cas d'utilisation. Néanmoins, un tel raffinement devrait être possible à d'autres niveaux de modélisation.

Les éléments d'un système possèdent le plus souvent des obligations mais aussi des attentes vis-à-vis d'autres éléments de ce système. En d'autres mots, ils dépendent les uns des autres. Cette dimension, appelée sociale, est très importante dans des systèmes qui se veulent coopératifs et dynamiques. Cependant UML ne permet pas de modéliser adéquatement cette dimension, et d'autres modèles tels que i* ou KAOS sont plus appropriés dans ce cas.
La méthodologie Tropos, basée sur i*, offre une approche différente d'UML mais complémentaire. La dimension sociale y est exprimée à l'aide du modèle de *dépendance stratégique* d'i* (ce sigle équivaut à « intentionnalité distribuée »).
Ce modèle insiste sur les dépendances sociales entre les acteurs présents qui dépendent d'autres acteurs pour satisfaire leurs buts ou recevoir leurs ressources. Cette dépendance peut être de quatre types, selon qu'elle est de but fonctionnel, non fonctionnel, de tâche ou de ressource. Utiliser le modèle de dépendance stratégique permet de mieux identifier les besoins et exigences du système et apporte une vision complémentaire à celle des cas d'utilisation. Par exemple, dans un tel modèle, on peut exprimer qu'un Four de cokerie dépend d'une Défourneuse pour réaliser le but fonctionnel Défournement (Figure 1).

Figure 1 : dépendance i*

Sans rentrer dans les détails de ce modèle, ajoutons néanmoins qu'un langage formel, appelé « Formal Tropos », basé sur KAOS, a été conçu pour compléter i*. Tout d'abord, il représente une notation textuelle pour les modèles i* et permet de décrire de manière linéaire les contraintes dynamiques entre les éléments d'un système. Ensuite, il est doté d'une sémantique définie de manière précise (ce qui fait défaut à UML comme nous l'avons vu plus haut), ce qui le rend accessible à une analyse formelle. Finalement, il peut aussi être soumis à des techniques de « model checking » qui peuvent être automatisées.

Dimension intentionnelle. Une autre limitation d'UML porte sur le manque d'expression du « pourquoi » du système et sur les restrictions au seul concept d'objet. Les acteurs présents et les associations entre les différents cas d'utilisation sont les principales données observables dans ce type de diagramme. Il serait cependant fortement utile d'exprimer les buts existants dans les différentes fonctionnalités qui sont effectuées.
Un deuxième type de diagramme i* permet de remédier à cette lacune : le modèle de *raisonnement stratégique*. Ce modèle est lui aussi exprimé en termes de buts, de tâches et de ressources. L'objectif de ce moyen de représentation est de décrire le raisonnement interne des acteurs, c'est-à-dire la manière dont les acteurs choisissent entre plusieurs alternatives. On utilise pour cela des liens fins-moyens et des décompositions de tâche, qui permettent de représenter la logique interne des tâches et de comprendre les liens qui les unissent.
Un *lien fins-moyens* exprime comment un but pourra être accompli au moyen de certaines tâches. Prenons l'exemple d'un but fonctionnel représenté à la Figure 2 : la Réception de charbon dans une cokerie. Celle-ci peut être accomplie par trois « moyens » différents, par chemin de fer, par l'eau ou par la route. Ces trois moyens permettent d'accomplir la fin, l'objectif, qui est la réception du charbon.

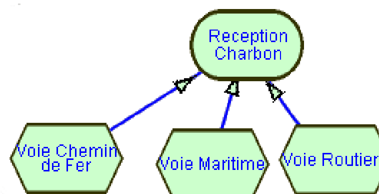

Figure 2 : liens fins-moyens

Une *décomposition des tâches* (Figure 3) exprime comment une tâche peut être décomposée en une série d'autres sous-tâches ou sous-buts. Il est ainsi possible de décomposer l'enfournement d'un four de cokerie en plusieurs sous-éléments. Ces différents buts et tâches seront eux même éventuellement décomposables ensuite en des éléments encore plus précis. Ils sont représentés sous forme d'arbre indiquant la nécessité de conjonction.

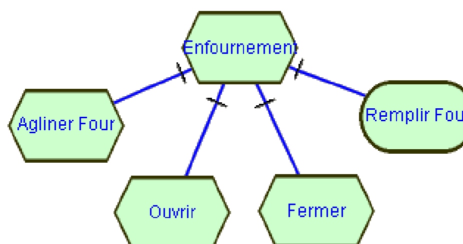

Figure 3 : décomposition de tâche

Utiliser des diagrammes i* en complément aux diagrammes UML est parfois une solution raisonnable. Cependant, il n'est pas évident de développer plusieurs modèles dans des langages différents, pour des raisons de cohérence ou pour des contraintes de temps. On pourrait envisager dans le futur une prise en compte par UML des avantages des modèles présentés. Un tel élargissement des horizons d'UML serait très souhaitable.

Agents. Un agent peut être défini comme un logiciel envoyé sur un réseau pour effectuer une tâche à la place de l'utilisateur et sans son intervention. Les agents sont utilisés par exemple pour le filtrage d'informations et la recherche sur Internet. De plus, un agent est dit intelligent lorsqu'il utilise les techniques de l'intelligence artificielle.
Les méthodologies de développement de logiciel orienté-agent gagnent en popularité, et ce au détriment des méthodologies orienté-objet. Les architectures basées sur les agents, aussi appelées systèmes multi-agents, permettent la création de systèmes ouverts, évolutifs, capable d'intégrer de nouveaux agents et de remplacer les moins performants. De plus les

systèmes multi-agents permettent de mieux gérer les circonstances imprévues.
On peut citer les méthodologies orienté-agent Gaia et Tropos. Cette dernière, que nous avons déjà rencontrée, utilise le cadre de modélisation i* pour le développement de logiciels orienté-agent. Contrairement à UML, l'approche Tropos est directement centrée sur les dimensions sociales et intentionnelles du système étudié. Cette caractéristique est très importante dans un environnement dynamique et multiple, comme par exemple celui des applications « e-business » téléchargeables qui doivent fonctionner sur toutes sortes de plateformes informatiques.
Parmi les avantages qu'offre Tropos par rapport à UML, citons la possibilité de garder des buts non opérationnalisés dans le modèle, empêchant ainsi de figer trop tôt la manière d'atteindre un objectif et laissant plus de liberté à une optimisation ultérieure lors du fonctionnement du système.

Cependant UML évolue lui aussi, et adopte à son tour de nouvelles fonctionnalités que nous allons à présent aborder.

Avenir d'UML. UML est aujourd'hui devenu un standard pour la description visuelle de la structure et du comportement des systèmes informatiques. C'est un outil en évolution permanente, dont la vie est jalonnée d'étapes. La version 2.0 d'UML est attendue pour le 30 avril 2004.
Cette seconde version d'UML devrait avoir un grand impact sur la modélisation orienté-objet, et sa principale caractéristique est d'élargir le cadre d'utilisation d'UML. Ce langage a en effet pour but ultime d'être un outil permettant la création de nouveaux logiciels, cependant, en se rapprochant d'initiatives actuelles comme le MDA (« Model Driven Architecture ») et le SOA (« Service-Oriented Architect »), UML 2.0 devrait évoluer pour décrire et automatiser les processus industriels, et devenir aussi un langage pour développer des systèmes indépendants de leur plateforme.
Un autre avantage d'UML 2.0 est qu'il permettra une meilleure modélisation de problèmes plus variés, en augmentant encore sa capacité d'adaptation. Ceci devrait être rendu possible par l'augmentation du nombre de diagrammes (treize au lieu de neuf actuellement), qui apporteront plus de souplesse à la modélisation.
Ces diagrammes sont répartis en deux nouvelles catégories : d'une part les *diagrammes de structure* qui décrivent les éléments d'un modèle qui ne dépendent pas du temps et d'autre part les *diagrammes comportementaux* qui décrivent les caractéristiques dynamiques d'un système ou d'un processus.

Il existe six types de diagrammes de structure, dont quatre sont déjà compris dans les versions UML 1.x :

- le diagramme de classe ;
- le diagramme d'objets ;
- le diagramme de composants ;
- le diagramme de déploiement.

Les nouveaux diagrammes sont :

- le « composite structure diagram », qui sert à décrire la structure interne d'un classifiant (comme par exemple une classe, un composant ou un cas d'utilisation), en incluant les points d'interactions de ce classifiant avec d'autres points du système ;
- le « package diagram », qui montre comment les éléments du modèle sont organisés en « package », et comment les « packages » dépendent les uns des autres.

En plus des diagrammes de structure, il existe sept diagrammes comportementaux, parmi lesquels cinq nous sont déjà familiers :

- le diagrammes de cas d'utilisation ;
- le diagramme d'activité ;
- le diagramme d'états-transitions ;
- le diagramme de séquence ;
- le diagramme de collaboration.

Les nouveaux diagrammes sont :

- l'« interaction overview diagram », qui est une variante du diagramme d'activité. Ce nouveau diagramme donne une vue sur les réseaux de contrôle au sein d'un système ou d'un processus industriel. Chaque nœud/activité dans le diagramme peut représenter un autre diagramme d'interaction ;
- le « timing diagram », qui décrit les changements d'état ou de condition d'un rôle ou d'une instance au fil du temps. Ce diagramme est typiquement utilisé pour montrer les changements d'état d'un objet en réponse à des événements extérieurs.

Notons que les diagrammes d'interaction forment maintenant un sous-groupe parmi les diagrammes comportementaux. Ils comprennent toujours les diagrammes de séquence et de collaboration, auxquels s'ajoutent le « timing diagram » et l'« interaction overview diagram ».
UML 2.0 n'a cependant pas conservé tels quels les diagrammes déjà utilisés dans les versions précédentes. Notons pour commencer que certains diagrammes ont changé de nom, tel le diagramme d'états-interactions désormais appelé « State Machine Diagram », et le diagramme de collaboration, renommé « Communication Diagram ». De plus, tous ont subi des modifications plus ou moins profondes, qui se traduisent le plus souvent par l'ajout de nouveaux concepts.
Cependant, l'étude de toutes ces nouvelles caractéristiques demanderait une place assez considérable, et déborderait largement du cadre de ce document.
Notons finalement l'existence d'une extension d'UML pour les agents, AUML. Cependant AUML n'est pas considéré aujourd'hui comme un standard, et il est principalement centré sur les protocoles d'interaction (comme les diagrammes de séquence) au détriment des autres diagrammes.