Ministère de l'Education Nationale, de la Recherche et de la Technologie

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

INSTITUT D'INFORMATIQUE D'ENTREPRISE

RAPPORT DE STAGE

BENTZ Cédric

Stage de 2e année

Directeur de stage : Franck MERTZWEILLER

Stage effectué à DATAPOINT

du 05/06/2001 au 31/08/2001

Table des matières :

I. ACCUEIL ET ENCADREMENT

J'ai effectué ce stage chez Datapoint, dans un service d'une dizaine de personnes, dirigé par William MASSONNEAU, directeur de projets. Ce service est installé dans les locaux de Bouygues Telecom à Vélizy. Mon directeur de stage, qui m'a encadré durant ces trois mois, était Franck MERTZWEILLER, ingénieur développement chez Datapoint. Etaient également présents :

- José BATISTA, consultant
- Pierre BREAU, ingénieur développement
- Damien COQUELLE, ingénieur développement
- Frédéric DELALANDRE, ingénieur développement
- Vincent HERNU, ingénieur développement
- Laurent LHEUR, ingénieur développement

J'ai aussi rencontré Bruno DELISLE, directeur technique, à plusieurs reprises.

Par ailleurs, j'ai également été encadré par Damien COQUELLE pour la première partie de mon stage portant sur le projet AMS, ainsi que par Frédéric DELALANDRE. Pour la troisième partie, portant sur l'API Java, j'ai été amené à travailler avec José BATISTA, ainsi que Laurent LHEUR.

REMERCIEMENTS :

Je remercie bien évidemment toutes les personnes de Datapoint détachées à Bouygues Telecom pour leur accueil chaleureux. Je remercie William de m'avoir permis d'effectuer mon stage dans son équipe, et surtout de m'avoir confié des projets motivants. Je remercie Franck, mon directeur de stage, pour son aide, son soutien, et sa bonne humeur. Je le remercie également de m'avoir mis en contact avec William. Je remercie Frédéric pour son aide et sa gentillesse. Je remercie Damien pour sa disponibilité, sa patience, et, malgré ce qu'en diront certains, pour son caractère agréable. Je remercie Laurent pour ses conseils précieux et sa gentillesse. Je remercie Pierre pour ses livres et sa bonne humeur. Je remercie José pour son aide, pour sa disponibilité et sa gentillesse. Enfin, je remercie également Vincent, Joss et Bruno, qui, même si je n'ai pas beaucoup eu l'occasion de les voir, ont été très sympathiques.

II. RESUME DU STAGE

Ce stage s'est composé de trois parties différentes. Lors de la première, j'ai travaillé sur le projet AMS, alors en cours de développement par les ingénieurs de Datapoint. Mon travail a consisté à adapter le projet pour que l'application aille récupérer ses paramètres de configuration, non plus dans un fichier (local), mais dans une base de données centralisée. Le développement a été effectué en langage C++. Une documentation résumant le format des données stockées a également été rédigée.

L'objectif de la deuxième partie était de définir la maquette d'une interface homme-machine (IHM) pour un tout nouveau projet de Datapoint, le projet URM2 (ce qui signifie Utilisation du Répondeur comme Média, 2^{e} version). L'accent a été mis sur l'ergonomie de cette IHM, qui a pour vocation d'être utilisée par de nombreux profils de personnes. L'étude des fonctionnalités de l'application finale a été déterminante pour faire les meilleurs choix graphiques possibles.

Enfin, la troisième et dernière partie de mon stage a été consacrée à la réalisation d'une API. Le langage utilisé a été le Java, et l'objectif était de définir puis d'implémenter un ensemble de classes destinées à faciliter la communication avec une application nommée CPI, en gérant de façon automatique l'envoi et la réception de messages. Les besoins utilisateurs imposaient quelques contraintes qui ont entrainé un certain nombre de problèmes à résoudre. Une documentation au format html, détaillant l'interface de chacune des trois classes élaborées, ainsi que le fonctionnement de l'API, a également été fournie.

III. PRESENTATION DE DATAPOINT

Le groupe Datapoint possède 11 filiales dans le monde entier et emploie plus de 700 personnes.
Son siège social se situe à Londres, et le chiffre d'affaire du groupe a été de cent trente trois millions de dollars en 1999. Ses filiales sont implantées dans 11 pays différents : Allemagne, Angleterre, Belgique, Espagne, Etats-Unis, France, Hollande, Italie, Norvège, Suède et Suisse.

La société Datapoint France, où a été effectué ce stage, compte actuellement 75 collaborateurs.
C'est une société anonyme (SA) au capital de cinq millions de francs, qui a été créée en 1976, et dont les locaux sont actuellement situés à l'Haÿ-les-Roses, dans le département du Val de Marne (94). Son chiffre d'affaire pour l'année fiscale 2000 a été de cent vingt trois millions de francs.

Son champ d'action s'étend à deux domaines d'activité :

- les centres de contact client : la société Datapoint réalise des centres d'appels permettant notamment de rationaliser et d'optimiser la gestion à distance de la relation avec la clientèle. A ce jour, plus de 500 sites ont déjà été mis en place par Datapoint.

- la gestion des médiathèques : la société Datapoint intervient dans ce domaine depuis 1983, et a équipé, à ce jour, plus de 300 bibliothèques. On peut citer notamment la médiathèque de la Cité des Sciences et de l'Industrie, la bibliothèque de l'Ecole Polytechnique, celle de l'Ecole Normale Supérieure de Lyon, etc.

Le stage s'est déroulé dans le service chargé de l'activité « centres d'appels ». Plus précisément, il a été réalisé au sein d'une équipe détachée à ***Bouygues Telecom*** et appartenant à cette branche de la société. En effet, Bouygues Telecom constitue un client important de la société Datapoint pour cette activité, et, pour cette raison, un groupe d'une dizaine de personnes travaille en permanence dans les locaux de Bouygues Telecom à Vélizy (78), où ils ont déjà réalisé de nombreux projets.

Grâce à la connaissance des technologies et des produits utilisés par Bouygues Telecom qu'ils ont acquise, ils sont devenus des prestataires dont l'expérience, atout précieux dans ce domaine, a permis par le passé et continue encore aujourd'hui d'assurer une collaboration fructueuse à de nombreuses occasions. De surcroît, la proximité des ingénieurs de Datapoint favorise la communication avec les équipes et les ingénieurs de Bouygues Telecom, facilite l'assistance et la formation aux nouveaux produits, et garantit l'insertion de cette équipe dans l'environnement de travail de Bouygues Telecom.

En outre, comme ce sont les ingénieurs de Datapoint qui possèdent le savoir-faire quant aux produits qu'ils ont déjà développés pour Bouygues Telecom, c'est tout naturellement à eux que les équipes de Bouygues Telecom font appel lorsqu'elles veulent faire évoluer ou bien améliorer une application existante et réalisée par Datapoint. Ainsi, par exemple, le projet AMS commandé à Datapoint fait suite au projet ATS réalisé quelques années plus tôt par les ingénieurs de Datapoint.

IV. L'ENVIRONNEMENT DU STAGE

1. L'ENVIRONNEMENT TECHNIQUE

L'environnement technique du stage a été relativement diversifié tout au long de ces trois mois. Le poste qui m'a été attribué tournait sous WINDOWS NT. La première partie du stage a été réalisée dans l'environnement de travail Visual Studio : le développement en C++ s'est fait sous Visual C++ (avec utilisation de Microsoft Source Safe), et le SGBD utilisé pour la base de données a été Oracle. J'ai également eu la chance de voir fonctionner un des ACD appartenant à Bouygues Telecom.

La deuxième partie, portant sur le projet URM2, a été réalisée, pour la maquette, sous Microsoft FrontPage.

Enfin, la troisième partie, qui consistait en la conception et l'implémentation de plusieurs classes Java, a été effectuée dans un environnement UNIX pour la première version, puis les classes ont été transférées sous WINDOWS NT où elles ont évolué jusqu'à la deuxième version.

2. L'ENVIRONNEMENT HUMAIN

Outre l'équipe de Datapoint que j'ai déjà détaillée, j'ai également été amené à rencontrer les équipes de production de Bouygues Telecom.

V. TRAVAIL REALISE

Comme je l'ai déjà précisé, mon stage s'est déroulé en trois parties. Mon plan traitera simplement chaque partie l'une après l'autre. Après avoir fait une brève présentation de l'architecture du centre d'appels Bouygues Telecom, nous verrons tout d'abord la partie rattachée au projet AMS, puis celle rattachée au projet URM2, et enfin nous examinerons la réalisation de l'API CPI.

1. L'ARCHITECTURE DU CENTRE D'APPELS DE BOUYGUES TELECOM

Le schéma correspondant est fourni en annexe (*Annexe 1*). Les appels clients (représentés par les lignes téléphoniques France Telecom) arrivent sur une machine nommée ACD, ou « Automatic Call Distributor ». Pour savoir sur quel CDC (conseiller de clientèle) l'appel doit être envoyé, l'ACD interroge, par l'intermédiaire d'un lien ASAI (ou « Adjunct Switch Application Interface »), un serveur, le T-Server (pour « Telephony Server », représenté par T-S sur le schéma), qui lui-même va interroger le routeur (IR). Ce dernier va interroger une base de données via le CPI, ou « Central Point of Information » (*voir l'annexe 6*), et choisir une « route » à partir des informations récupérées. Puis, il va en informer le T-Server, qui va, à son tour, transmettre l'information à l'ACD. Ainsi, en fonction du client (identifié au niveau de l'ACD par son numéro de téléphone), on choisit un CDC disponible et possédant les compétences appropriées. De surcroît, le T-Server fournit au poste du CDC les informations concernant le client dont l'appel est traité. On appelle CTI (pour « Couplage Téléphonie-Informatique ») toute la partie informatique de l'architecture, c'est-à-dire dans le cas de Bouygues Telecom, l'ensemble constitué par le T-Server et le routeur.

2. LE PROJET AMS

Il y a quelques années, les ingénieurs de Datapoint avaient réalisé une application du nom de « ATS », ou « Agent Telephony Screen ». Pour résumer, disons que cette application consiste en une barre téléphonique qui apparaît sur l'écran du CDC, par laquelle ce dernier va décrocher, raccrocher, ou même transférer les appels des clients, recevoir des informations concernant l'appel, etc. Tout ce qui concerne son travail se fait entièrement par l'intermédiaire de l'ATS, et non plus directement par le poste téléphonique. Cela permet le « free-sitting », c'est-à-dire qu'un CDC peut s'installer là où il veut. Il suffit qu'il se logue à son nom sur la machine où il travaille, et ainsi les appels qui doivent être traités par lui seront convenablement dirigés vers son poste. Les paramètres de configuration de la barre ATS (qui incluent des données telles que « la fenêtre qui va apparaître à l'écran lors d'un appel a telle couleur, telle taille, et délivre telles informations ») sont stockés dans un fichier local sur le poste NT de chaque CDC.

Le projet « AMS » est une évolution du projet ATS. La tâche qui m'a été confiée, et qui a constitué la première partie de mon stage, était de modifier le projet AMS en cours pour que les informations concernant la configuration de l'application ATS sur chaque poste soient récupérées, non plus dans un fichier d'initialisation local au poste (ats.ini), mais dans une base de données centralisée (où les informations sont donc accessibles via le

réseau à tous les postes), dans une table nommée « Config ». Le projet était écrit en C++ et réalisé sous Visual C++.

a. Les différentes sections du fichier « ats.ini »

Le fichier est organisé à la façon d'un arbre : la racine du fichier est « sections ». Ensuite, le deuxième niveau de l'arbre est l'ensemble des sections : « MAIL », « GUI », « COM », « CUSTOM », « TSERVER », « ENGINE », « OgmTabDlg », « WINDOWS », « TEVENTS » et « DavoxMessages ». Le troisième niveau est différent selon les sections. Pour « MAIL », « GUI », « COM », « CUSTOM », « TSERVER », « ENGINE » et « OgmTabDlg », il est constitué par les clés de chacune des sections : « Log » pour la section « COM » ; « Log », « Big Button », etc. pour la section « GUI » ; « Override », « CPI », etc. pour la section « CUSTOM » ; etc. En face de ces clés se trouve la valeur de l'association section/clé. Nous verrons plus tard comment sont organisées les sections « WINDOWS », « TEVENTS » et « DavoxMessages ».

Par exemple, on aura dans le fichier la structure d'arbre à quatre niveaux suivante :

[racine] [section] [clé] [valeur de l'association section/clé]
[niveau 1] [niveau 2] [niveau 3] [niveau 4]

sections –
|GUI –
| |Log……« Y »
| |Big Button… « 0 »
|
|COM –
| |Log……« Y »
|
|CUSTOM –
| |Override……« N »
| |CPI……« on »
|
|*etc.*

b. La classe CSGBDConfig

Le premier algorithme implémenté avait pour but d'adapter la récupération des informations liées aux sections « MAIL », « GUI », « COM », « CUSTOM », « TSERVER », « ENGINE » et « OgmTabDlg ». Cette récupération était auparavant effectuée en « parsant » le fichier « ats.ini », et le nouvel algorithme devait faire en sorte qu'elle le soit à présent en utilisant la table « Config ».

La première étape a été de réfléchir à la manière dont les informations allaient être stockées dans la base. La solution retenue est simple : on garde la structure d'arbre,

en stockant les informations sous forme clé/valeur, en utilisant les virgules comme séparateurs. Pour les cas où on aurait des clés multiples au même niveau de l'arbre (par exemple, « Log » existe à la fois dans « COM » et dans « GUI »), on utilise des numéros. Ainsi, pour différencier « Log » qui dépend de la section « COM » et « Log » qui dépend de la section « GUI », on appelle le premier « Log1 » et le deuxième « Log2 ». Tout cela est réalisé dans le but d'obtenir des clés uniques, car c'est une condition nécessaire à la bonne marche de l'algorithme.

On aura donc :

[clé] *[valeur]*

sections MAIL, COM, GUI, CUSTOM, TSERVER, ENGINE, OgmTabDlg, WINDOWS, TEVENTS, DavoxMessages
COM Log1
GUI Log2, BigButton
CUSTOM Override, CPI
Log1 Y
Log2 Y
BigButton 0
Override N
CPI on
etc.

L'arbre se présente donc sous la forme d'une grammaire non récursive. En effet, il est équivalent à :
sections = MAIL|COM|GUI|CUSTOM|TSERVER|ENGINE|OgmTabDlg, COM = Log1, GUI = Log2|BigButton, CUSTOM = Override|CPI, Log1 = Y, Log2 = Y, BigButton = 0, Override = N, CPI = on, etc.

L'étape suivante, pour exploiter les données de l'arbre, a été de créer et d'implémenter la classe **CSGBDConfig**.

La classe **CSGBDConfig** possède deux méthodes fondamentales : la méthode ***Init*** et la méthode ***get_ProfileString***. La première consiste en fait à récupérer toutes les données de la table « Config » qui contiennent les informations relatives aux paramètres de configuration du poste (car la table « Config » possède aussi des informations inutiles à l'algorithme). En fait, il est intéressant de noter que, pour stocker les données dans la base, un préfixe est ajouté devant le nom de chaque clé : ici, il s'agira du préfixe « conf_ ». On aura donc : conf_sections, conf_COM, conf_GUI, etc. Ce préfixe est utilisé dans la base pour indiquer clairement à quel groupe de données appartient l'information, ou plutôt le tuple (ici, il s'agit donc des informations concernant les paramètres généraux de configuration du poste). De la même façon, le préfixe « wnd_ » sera utilisé pour les informations concernant les paramètres des fenêtres (« windows ») qui vont apparaître lorsque l'application sera exécutée sur le poste ; et « dvx_ » pour les informations concernant les messages DAVOX. Les seuls tuples dont a besoin la classe CSGBDConfig sont donc ceux dont la clé commence par « conf_ ». Une fois que toutes les données de la table « Config » sont récupérées dans la base (ce qui se fait en dehors de la classe

CSGBDConfig), la méthode ***Init*** peut être appelée. Cette méthode prend en argument une instance de la classe **CData** : il s'agit d'une classe créée par l'équipe de développement de Datapoint, qui permet entre autres (et c'est la propriété que l'on va utiliser ici) de stocker des données sous forme d'un ensemble de couples (« clé », « valeur »). On passe en argument de la méthode ***Init*** une instance de la classe **CData** contenant uniquement les informations utiles à l'algorithme, et elle la recopie une fois pour toutes dans une **CData** locale : ainsi, une fois que la méthode ***Init*** a été appelée, la deuxième méthode, ***get_ProfileString***, dont nous allons à présent étudier le fonctionnement, peut être appelée autant de fois que nécessaire.

En local, on a donc les informations sous la forme : CDatalocale = ((« conf_sections », « MAIL, COM, GUI, CUSTOM, TSERVER, ENGINE, OgmTabDlg »), (« conf_COM », « Log1 »), (« conf_GUI », « Log2,BigButton »), …, (« Log1 », « Y »), *etc.*)
C'est à partir de cette structure de données que l'on va travailler. La seconde méthode, ***get_ProfileString***, utilise la **CData** locale initialisée et va chercher les informations qui lui sont demandées. Elle prend comme paramètres deux chaînes de caractères, qui correspondent respectivement à une section et à une clé, et retourne la valeur associée à ce couple. Par exemple, ***get_ProfileString*** (COM, Log1) renvoie la valeur « Y ». Si la section n'existe pas, elle renvoie « erreur de section » ; si la clé n'existe pas, elle renvoie « erreur de clé ». Dans cet algorithme, on utilisera la fonction ***Tokenize***, développée par l'équipe de Datapoint, qui permet de décomposer une chaîne de caractères en précisant le séparateur, et de stocker les sous-chaînes obtenues dans un vecteur. Par exemple, si on applique ***Tokenize*** à la chaîne string = « string1,string2 » en précisant d'utiliser la virgule comme séparateur, le résultat renvoyé sera le vecteur vecteur_résultat = | string1 | string2 |.

Détail du fonctionnement de l'algorithme (les arguments sont sectionName et keyName) :

trouvé = non
Récupérer et décomposer en noms de sections la valeur associée à la clé « conf_sections »

Tant qu'il reste des sections et que trouvé = non
    Si le nom de section courant = sectionName alors
        trouvé = oui
    FinSi
FinTantque

Si trouvé = oui alors
    Récupérer et décomposer en noms de clés la valeur associée à la clé « conf_sectionName »
    trouvé = non

    Tant qu'il reste des clés et que trouvé = non
        Si le nom de clé courant = keyName alors
            trouvé = oui

```
                FinSi
        FinTantque

    Si trouvé = oui alors
                Retourner la valeur associée à la clé keyName
        Sinon
                Retourner « erreur de clé »
        FinSi
Sinon
        Retourner « erreur de section »
FinSi
```

Le listing détaillé de la classe CSGBDConfig est fourni dans l'annexe 2.

c. La section « WINDOWS »

La deuxième modification réalisée avait, quant à elle, pour but d'adapter la récupération des informations liées à la section WINDOWS, section qui contient la description des fenêtres utilisées par l'application. On trouve des informations générales, comme la taille de la fenêtre, son titre, sa couleur, mais également des renseignements concernant les champs qui la composent (« Controls »). Ces informations sont stockées dans le fichier sous la forme suivante :

[fenêtres] *[caractéristiques de la fenêtre]* *[caractéristiques des [valeurs]* *« Controls » associés]*

```
window1

    Dim…….50,50,50,50
    Caption……Window1
ForeColor……0
    Control1
                    Name……C1W1
                Type……..AtsLabel
                Dim……..40,40,40,40
                Caption…….ControlI1
                        etc.
Control2
                        Name…….C2W1
                        Type……..AtsFrame
                                Dim……..30,30,30,30
                                Caption…….....ControlI2
                        etc.
  etc.

window2

    Dim…….80,80,80,80
            Caption……
Window2
            ForeColor……0
            Control1
```

Name…….C1W2
Type…….AtsLabel

Dim…….70,70,70,70
Caption……ControlII1

etc.

Control2

Name……C2W2
Type……
AtsFrame

Dim…….60,60,60,60

Caption……
ControlII2

etc.

etc.

etc.

Le nombre de « Controls » d'une fenêtre (« window ») n'est pas limité. Encore une fois, la première étape a été de réfléchir à une façon de stocker ces informations dans la table « Config ». D'abord, on peut remarquer pourquoi la section « WINDOWS » ne peut pas être traitée avec la même structure de données que précédemment : auparavant, les valeurs (les feuilles de l'arbre) se trouvaient toujours au même niveau de l'arbre, le quatrième. Ici, comme on peut le voir, certaines se situent au quatrième niveau et d'autres au cinquième (n'oublions pas que le premier niveau de l'arbre est la racine « sections », non représentée ici). Les fenêtres sont parcourues une par une, et lorsque l'une d'entre elles a fini d'être parcourue, on veut que toutes les informations la concernant soient connues, pour qu'elle puisse ensuite être créée (c'est le but final de l'opération) : on va d'abord renseigner les champs généraux de la fenêtre, puis on parcourt les « Controls » associés à la fenêtre pour les créer un par un. Enfin, on fournira à la fenêtre cette liste de « Controls ». On distingue donc deux niveaux différents dans l'arbre : celui des caractéristiques générales des fenêtres, et celui des caractéristiques de leurs « Controls » respectifs. Pour ne pas trop augmenter la taille de la base, on a décidé de limiter le nombre de tuples. Ainsi, les caractéristiques des fenêtres seront renseignées sur une seule ligne, séparées par « \\ » (et avec un séparateur de champ spécifique pour séparer la caractéristique de sa valeur, qui est « -> »), alors que celles des « Controls » le seront au niveau suivant. Rappelons également que tous les tuples concernant les fenêtres (et donc également les « Controls ») ont leur clé qui commence par « wnd_ ». Enfin, encore une fois, les clés doivent être uniques dans la table « Config », et les « Controls » seront donc numérotés : le premier numéro, sous forme de chiffre romain (I,II,III,etc.), est le numéro de la fenêtre à laquelle le « Control » est rattaché, et le deuxième est le numéro du « Control » au sein de sa fenêtre.

Pour résumer, on a donc :

[clé]	*[valeur]*
wnd_WINDOWS……..	window1,window2,*etc.*
wnd_window1……	Dim->50,50,50,50\\Caption->Window1\\ForeColor->0*etc.*\\ControlI1\\ControlI2*etc.*

wnd_ControlI1……...Name->C1W1\\Type->AtsLabel\\Dim->40,40,40,40\\Caption->ControlI1
wnd_ControlI2……...Name->C2W1\\Type->AtsFrame\\Dim->30,30,30,30\\Caption->ControlI2
wnd_window2……Dim->80,80,80,80\\Caption->Window2\\ForeColor->0*etc.*\\ControlII1\\ControlII2*etc.*
wnd_ControlII1……..Name->C1W2\\Type->AtsLabel\\Dim->70,70,70,70\\Caption->ControlII1
wnd_ControlII2……..Name->C2W2\\Type->AtsFrame\\Dim->60,60,60,60\\Caption->ControlII2

etc.

Après cette étape préliminaire, il restait à créer la classe **CWndParserDb** et à implémenter les méthodes permettant d'exploiter les données concernant les fenêtres (« windows ») et les « Controls ». Elles consistent encore une fois à effectuer un « parsing » des données récupérées dans la base au format décrit ci-dessus, c'est-à-dire à parcourir ces données en inscrivant les informations trouvées dans les variables nécessaires à la classe **CWindowManager** pour créer les fenêtres.

Sans entrer dans les détails, il faut garder à l'esprit que les informations à récupérer sont différentes selon le type du « Control » « parsé », et chaque type constitue donc un cas à part. L'implémentation de la classe **CWndParserDb** ne sera pas détaillée, mais un extrait du listing est fourni en annexe (*Annexe 3*).

d. La section « TEVENTS »

La troisième modification réalisée avait pour but d'adapter la récupération des informations liées à la section TEVENTS, qui contient la description des événements associés à l'application. L'organisation de cette section est très simple, je vais l'expliquer brièvement. Elle consiste en fait en un ensemble de couples conditions-actions (ce sont les « profiles »), qui s'interprètent de la façon suivante : si les conditions associées à une liste d'actions s'avèrent réalisées, alors ces actions sont exécutées.

Dans le fichier, on a donc une suite d'enchaînements du type :

	[variable]	*[valeur exigée]*
profile1		
	Event	ESTABLISHED
	APPLI	NOMAD
	conf_D_GOP	Y

Ici, on a la liste des conditions qui doivent être vérifiées.

	[action]	*[paramètres]*

actions
Initiate GOP_DDEServer, GOP_CA
Execute GOP_DDEServer, GOP_CA, DDE_OS, Parameters {MSISDN, LIBCT}

Ici, on a la liste des actions à exécuter si toutes les conditions sont vraies.

L'extrait de fichier ci-dessus signifie donc : si la variable *Event* a comme valeur 'ESTABLISHED' et que la variable *APPLI* a comme valeur 'NOMAD' et que la variable *conf_D_GOP* a comme valeur 'Y', alors les deux actions ***Initiate*** et ***Execute*** vont être exécutées, avec comme arguments *GOP_DDEServer* et *GOP_CA* pour ***Initiate***, et *GOP_DDEServer*, *GOP_CA*, *DDE_OS* et *Parameters {MSISDN, LIBCT}* pour ***Execute***.

La façon dont les données ont été stockées dans la base résulte de plusieurs contraintes. On aurait pu définir les « profiles » de la même façon que l'on a défini les « windows », c'est-à-dire en utilisant uniquement des séparateurs de champs, quitte à stocker toutes les informations sur la même ligne. Cependant, il aurait fallu ajouter un autre séparateur pour différencier les conditions des actions, et encore un autre pour lister tous les paramètres contenus entre accolades après le mot-clef *Parameters* (car *Parameters {param1, param2, etc.}* est en fait une liste de paramètres interne à la liste des paramètres de l'action ***Execute***. Dans l'exemple ci-dessus, les deux éléments de la liste de paramètres *Parameters* sont *MSISDN* et *LIBCT*.) Ce qui aurait fait trois séparateurs différents sur la même ligne ! On a préféré n'en garder que deux pour des raisons de lisibilité, quitte à augmenter un peu la taille de la base. On aurait même pu détailler les différents éléments de la liste *Parameters* dans un autre tuple, mais la solution intermédiaire paraissait la plus satisfaisante : les tuples obtenus sont encore lisibles et la taille de la base reste raisonnable.

La solution retenue est donc la suivante : chaque « profile » est défini avec le nom de sa liste de conditions et le nom de sa liste d'actions.
Ensuite, pour détailler les conditions et les actions, les séparateurs seront toujours « \\ » entre deux actions ou entre deux conditions, « -> » entre une rubrique et sa valeur (ou ses paramètres), et on ajoute le signe « |* » pour séparer les paramètres listés dans la rubrique *Parameters*. Remarquons enfin que, comme les actions sont associées aux fenêtres (« windows »), chaque tuple concernant la section « TEVENTS » a sa clé qui commence par le préfixe « wnd_ ».

Pour résumer, on a donc :

[clé] *[valeur]*

wnd_TEVENTS…….profile1,profile2,*etc.*
wnd_profile1……..Conditions1,Actions1
wnd_Conditions1……...Event->ESTABLISHED\\APPLI->NOMAD

wnd_Actions1……Hide->window1\\Show->window3
wnd_profile2……..Conditions2,Actions2
wnd_Conditions2……...Event->ESTABLISHED
wnd_Actions2……Execute->Server,Test,DDE_OpenBlankScreen, Parameters->MSISDN|*LIBCT\\Show->window1
etc.

Il reste à écrire la classe **CTEventsWndParserDb**, qui permet de « parser » cette structure. On implémente deux méthodes principales : celle effectuant le « parsing » des actions, et l'autre effectuant le « parsing » des conditions. Pour chaque « profile », c'est-à-dire chaque couple associant une liste de conditions à une liste d'actions, on parcourt les conditions et les actions associées. Encore une fois, l'implémentation complète de la classe ne sera pas détaillée, mais un extrait du listing est fourni en annexe (*Annexe 4*).

e. La section « DavoxMessages »

Pour cette section, il n'a pas été nécessaire de réimplémenter une classe, il a suffit d'ajouter une méthode ***InitFromDb*** à la classe **CMessageFactory**. Cette section contient la description de messages utilisés par l'application. Plus précisément, elle contient la description du format de ces messages. Ainsi, elle se présente sous la forme d'une liste de messages, chacun composé de plusieurs champs. En face du nom de chaque champ se trouve donc sa description, sur quatre colonnes : la première indique si c'est un champ simple (S) ou non, la seconde indique sa longueur en nombre de caractères, et la troisième et la quatrième indiquent respectivement l'alignement des données (aucun, à droite, à gauche) et le caractère de remplissage pour l'alignement (caractère vide par défaut).

Tout d'abord, voyons à quoi ressemble cette section dans le fichier de configuration local :

[identifiant du message]	*[intitulé du champ]*	*[description du champ sur 4 colonnes]*			
<901					
	trash	S	3	-	-
	MSISDN	S	10	-	-
	UserData2	S	55	-	-
	UserData3	S	40	-	-
	trash	S	4	-	-
	UserData4	S	35	-	-
	UserData5	S	45	-	-

<902

trash	S	5	-	-
UserData1	S	30	-	-
UserData2	S	40	-	-

<903

trash	S	7	-	-
UserData2	S	50	-	-

Pour stocker ces données dans la base, la structure simple en arbre semblait à nouveau tout à fait appropriée, et sans même que l'on ait besoin d'utiliser des séparateurs de champs supplémentaires. Pourtant, encore une fois, c'est-à-dire comme lors de l'implémentation de la classe **CSGBDConfig**, s'est posé le problème de l'unicité des clés : en effet, on remarque que, pour un même message, on peut avoir plusieurs champs de même nom (cf. le message d'identifiant **<901**, qui possède deux champs « trash »). A nouveau, la solution retenue a été d'ajouter des numéros, avec la même convention que celle utilisée pour la numérotation des « Controls » de chaque fenêtre : on ajoute, à la fin du nom du champ, le numéro du message auquel il appartient (en chiffres romains), et le numéro de ce champ dans le message.

Enfin, rappelons que le préfixe approprié concernant tout ce qui se rapporte à la section « DavoxMessages » est « dvx_ ».

Dans la base, on aura donc :

[clé]	*[valeur]*
dvx_DavoxMessages	<901,<902, <903
dvx_<901	trashI1,MSISDNI1,UserDataI2,UserDataI3,trashI2
dvx_<902	trashII1,UserDataII1,UserDataII2
dvx_<903	trashIII1,UserDataIII2
dvx_trashI1	S,3,-,-
dvx_MSISDNI1	S,10,-,-
dvx_UserDataI2	S,55,-,-
dvx_UserDataI3	S,40,-,-
dvx_trashI2	S,4,-,-
dvx_trashII1	S,5,-,-
dvx_UserDataII1	S,30,-,-
dvx_UserDataII2	S,40,-,-
dvx_trashIII1	S,7,-,-
dvx_UserDataIII2	S,50,-,-

A présent que l'on a vu la façon dont les sections avaient été stockées dans la base et exploitées, il reste à signaler que l'implémentation de ces classes n'aurait pas eu grand

intérêt si la classe **CWindowManager**, qui s'occupe de la gestion des fenêtres, n'avait pas été modifiée en conséquence. Je ne détaillerai pas tous les changements qui y ont été opérés, dans la mesure, où, encore une fois, il n'a s'agit que d'une adaptation du code existant. Soulignons quand même que d'autres classes, comme **CAtsWnd** et **CTEventWnd**, ont également dû être adaptées : il a s'agit principalement de créer et d'implémenter une ou deux méthode(s) supplémentaire(s) pour chaque classe.

Au final, et en conclusion de cette partie, toutes les modifications et adaptations nécessaires ont donc été réalisées : l'application va à présent chercher toutes les informations de configuration dont elle a besoin dans la table « Config ». Le principal avantage de cette réalisation est qu'il pallie un des inconvénients de la version précédente : auparavant, si l'on souhaitait modifier ou faire évoluer les paramètres de configuration d'un certain profil de postes, il fallait rééditer un nouveau fichier « ats.ini », et le télédistribuer à tous les postes concernés, ce qui est problématique lorsque le parc infomatique total est de 3000 postes répartis sur 6 sites en France.

Désormais, il suffit simplement de modifier le profil directement dans la base : ainsi, à chaque lancement de l'application, c'est la dernière version de la base (donc celle qui contient les paramètres de configuration mis à jour) qui sera chargée et utilisée par l'application.

A présent, nous allons étudier le travail relatif au projet URM2, réalisé dans la deuxième partie du stage.

1. LE PROJET URM2

Le projet URM (pour « Utilisation du Répondeur comme Média »), réalisé il y a quelques années par les ingénieurs de Datapoint, avait pour objet de permettre la gestion de vastes campagnes de communication. Le principe d'une telle campagne est de déposer des messages vocaux sur les répondeurs de personnes appartenant à un ensemble cible prédéfini. Le projet URM2, quant à lui, est une évolution d'URM : son objectif est, tout en reprenant l'intégralité des fonctionnalités d'URM, de permettre en outre une meilleure planification des campagnes (grâce à un moteur de planification), d'optimiser l'utilisation des machines chargées de l'envoi des messages en masse, et de fournir une interface homme-machine (IHM) agréable et facile d'utilisation pour permettre une gestion plus aisée et plus automatisée des campagnes. En effet, ce sont les commerciaux qui décident de lancer telle ou telle campagne de promotion, et ils doivent donc disposer d'outils qui facilitent efficacement leur travail.

La tâche qui m'a été confiée sur URM2 était de définir l'aspect et les fonctionnalités de l'IHM du projet. Le point le plus important était de proposer l'IHM la plus agréable possible d'un point de vue graphique et ergonomique, et qui permette une navigation et un enchaînement de pages les plus intuitifs possibles. Ce qu'il faut savoir concernant cette IHM, c'est qu'elle peut être utilisée par plusieurs profils d'utilisateurs, chacun d'entre eux ayant accès à des fonctionnalités différentes. Par exemple, le profil d'administrateur donne accès à tous les droits sur les campagnes.

Un administrateur peut donc démarrer, arrêter ou supprimer n'importe quelle campagne. Il peut aussi décider d'en créer une. Le super-utilisateur, quant à lui, est le chef de son service. Il a donc accès à tous les droits concernant les campagnes créées par son service, mais n'a que les droits en lecture pour les campagnes créées par un autre service. Par exemple, il peut décider de modifier une campagne créée par son service, mais n'a que le droit de consulter celles créées par d'autres services, et en aucun cas il ne peut les modifier. Enfin, l'utilisateur a seulement le droit de consulter les campagnes de son service. Il ne peut modifier ni créer aucune campagne. Pour en revenir à l'IHM, cela signifie concrètement que l'utilisateur ne devrait pas se voir proposer "créer une campagne" lorsqu'il l'utilise. L'affichage à l'écran dépend donc du profil de l'utilisateur connecté.

Pour créer une campagne, plusieurs paramètres doivent être précisés :

- le nom/l'identifiant de la campagne,
- le message vocal déposé dans les répondeurs,
- le fichier contenant les numéros de téléphone des personnes ciblées par la campagne,
- les plages horaires où doivent être effectués les envois de messages.

Il existe d'autres paramètres, mais ce sont les principaux. Les commandes permises par l'interface doivent être :

- créer une campagne,
- modifier une campagne,
- suspendre une campagne,
- arrêter une campagne,
- supprimer une campagne,
- planifier/réserver une campagne,
- consulter une ou plusieurs campagnes,
- ajouter/supprimer/modifier un utilisateur.

Pour ce faire, on a choisi de regrouper ces actions en 4 « rubriques » (elles-mêmes séparées en sous-rubriques) :

- la première concerne la gestion des campagnes : création, modification, suspension, arrêt, suppression.

- la deuxième concerne la réservation/planification d'une campagne : c'est une rubrique à part entière car réserver/planifier une campagne est une tâche complexe, qui se déroule en plusieurs étapes. Il faut d'abord réserver la campagne, c'est-à-dire proposer des plages horaires, puis interroger le moteur de planification qui peut exiger certains changements, et, lorsqu'un créneau est approuvé par le moteur, il faut planifier la campagne, c'est-à-dire bloquer définitivement le créneau.

- la troisième concerne la consultation des campagnes : cela suppose de pouvoir consulter une campagne en particulier, ou de consulter un certain nombre d'attributs des campagnes d'un même service (ces attributs étant paramétrables), etc.

- la quatrième concerne la gestion des utilisateurs (rubrique non accessible aux utilisateurs simples).

L'accès à ces quatre rubriques s'effectue par des liens contenus dans un cadre vertical placé à gauche de l'écran, et représentés chacun par une image. Ainsi, à chaque instant, l'utilisateur peut accéder à n'importe quelle rubrique. Ensuite, un cadre horizontal placé en bas de l'écran fournit des indications sur le fonctionnement du système, également accessibles à tout moment : on y trouve un lien pour écrire à l'administrateur en cas de besoin, une aide en ligne, et même une icône qui informe l'utilisateur des alertes remontées par le système en cas de problème. De plus, un cadre horizontal placé en haut de l'écran contient le logo de Bouygues Telecom et le nom de la sous-rubrique courante. Enfin, tout en haut du cadre placé au centre de l'écran se trouvent des onglets qui permettent d'accéder à toutes les sous-rubriques de la rubrique courante. L'ensemble de l'IHM est réalisé aux couleurs de Bouygues Telecom. Une capture d'écran montrant la maquette réalisée est fournie dans l'annexe 5.

1. L'API CPI

a. Le CPI

Le CPI permet de faire communiquer plusieurs applications entre elles. Pour chaque application désirant se connecter au CPI, ce dernier crée un « DIP », ou « Device Interface Procedure ». A une application est donc associée un et un seul DIP : une fois qu'une application possède un DIP, c'est-à-dire une fois qu'elle est connectée au CPI, elle peut communiquer avec toutes les autres applications également connectées, en passant par le CPI. L'annexe 6 illustre le fonctionnement du CPI : le routeur (IR) est connecté au CPI par le DIP « I » et l'IACCPI par le DIP « B ». Ainsi, si l'IR veut interroger la base de données (BD), il interroge, par l'intermédiaire du CPI, l'IACCPI, qui se chargera d'interroger la base et enverra la réponse à l'IR, toujours par l'intermédiaire du CPI. Le CPI communique avec les applications connectées à l'aide de messages à un format spécifique. Il faut noter que l'on peut également communiquer directement avec un des IACCPI chargé d'interroger la base « 24/24 » (base des clients possédant un forfait), ou avec un des AACCPI chargé d'interroger la base « Remus » (base des clients possédant une carte prépayée), sans passer par le CPI, mais en respectant le même format de messages. D'ailleurs, l'utilisation du CPI pour la communication entre les applications est amenée à disparaître, et pouvoir communiquer directement avec un AACCPI ou un IACCPI est donc une fonctionnalité importante.

b. Description de l'API

Une API (Application Programming Interface) est un ensemble de classes conçu pour automatiser et faciliter la réalisation d'une tâche particulière, par l'utilisation des méthodes définies dans l'interface de ces classes. L'objectif de cette partie de mon stage était de concevoir et de développer, en langage **Java**, une API destinée à permettre la communication avec le CPI : la gestion de l'envoi des requêtes au CPI ainsi que la réception des messages en provenance du CPI devaient être entièrement gérées. En outre, cette API devait pouvoir être utilisée pour communiquer directement avec un (ou plusieurs) IACCPI (ou « Interface Automatic Call Distributor & Central Point Information »), ou un (ou plusieurs) AACCPI. Ainsi, par la suite, tout ce que je dirai concernant le CPI s'appliquera également dans le cas d'un IACCPI ou d'un AACCPI, sauf précision contraire.

Les besoins des utilisateurs étaient :

- une gestion automatisée de l'envoi et de la réception des messages CPI.
- un fonctionnement possible et sécurisé même en cas d'utilisation simultanée de l'API par plusieurs « threads ».
- l'existence d'une et une seule instance (même si cette dernière peut être utilisée par plusieurs « threads » en même temps) de la classe qui se charge d'effectuer les requêtes : la classe mise à disposition de l'utilisateur doit donc être une classe « singleton ».
- une documentation précise mais concise, détaillant le fonctionnement et l'utilisation de l'API.

Avant d'examiner plus en détail la conception de l'API, rappelons rapidement ce qu'est un « thread ». Un « thread » est un objet de la classe **Thread**, chargé d'effectuer une tâche particulière. La principale caractéristique des « threads » est qu'ils sont susceptibles de partager des ressources : il faut donc faire très attention à la façon dont les ressources sont utilisées, pour éviter les conflits. Il ne faudrait pas, par exemple, qu'un « thread » accède en lecture à une variable partagée pendant qu'un autre la modifie. Pour éviter ce genre de problèmes, on verrouille ces variables : ainsi, lorsqu'une telle variable est utilisée par un « thread », elle devient inaccessible aux autres « threads » tant que ce dernier ne la libère pas. Ce phénomène est comparable à l'utilisation de sémaphores : lorsqu'un processus accède à une ressource, il exécute un « P » qui en bloque l'accès pour les autres processus, et une fois qu'il n'a plus besoin de cette ressource, il la libère grâce à un « V ». D'un autre côté, l'utilisation de tels mécanismes peut être à l'origine de phénomènes d'interblocage : si un processus p1 détient une ressource A et attend de pouvoir accéder à une ressource B pour continuer son exécution et que, d'autre part, un autre processus p2 détient la ressource B et attend la ressource A pour pouvoir la libérer, alors il y a interblocage, car les deux processus sont bloqués indéfiniment, chacun détenant la ressource nécessaire à l'autre. Il faut donc vérifier qu'on a « exclusion mutuelle ».

Pour l'application que j'ai eue à développer ici, le travail principal a plutôt été de rendre l'application « thread-safe » (c'est-à-dire utilisable sans danger par plusieurs « threads » simultanément) que de gérer l'exclusion mutuelle, qui, dans notre cas ne pose pas de problème particulier. Pour gérer la programmation « multi-threads » en Java, on utilise des méthodes synchronisées : une méthode synchronisée représente une ressource partagée protégée par un sémaphore. Lorsqu'un « thread » appelle une des méthodes synchronisées d'un objet, celui-ci devient verrouillé, et seules ses méthodes non synchronisées peuvent alors être appelées par d'autres « threads » tant qu'il est verrouillé.

Avant d'étudier ces problèmes spécifiques à l'utilisation de l'API par plusieurs « threads », voyons par quel moyen notre application communique avec le CPI. La première étape est évidemment d'établir une connexion réseau avec le serveur à atteindre : cela se fait tout naturellement en Java par la création d'une « socket » (c'est-à-dire d'un objet de la classe **Socket**), et la communication se déroule alors selon le protocole TCP/IP. C'est alors qu'intervient la gestion automatisée d'envoi de messages au CPI, ainsi que la réception des réponses associées. En effet, la particularité de la communication avec le CPI est qu'elle se fait par l'intermédiaire de messages possédant un format spécifique. Chaque message envoyé au CPI (qui est le seul type de message compréhensible par lui), et de même chaque réponse en provenance du CPI, se présente sous la forme d'une chaîne de caractères de longueur maximale 1024, constituée :

- d'un en-tête de longueur fixe (48 caractères),
- du corps du message proprement dit, d'une longueur maximale de 976 caractères.

Ce format de message est appelé le format « transactionnel ». Pour l'application que je devais réaliser, seul le format de l'en-tête m'intéressait. Concernant le corps du message, on considère simplement que la chaîne qui le constitue est récupérée d'une

autre application et qu'on l'inclut telle quelle dans le message. Par contre, il est nécessaire d'examiner le format exact de l'en-tête.

Composition de l'en-tête :

- Le premier caractère représente l'adresse de l'application destinataire du message : par exemple, 'A' pour désigner l'application A.

- Les deuxième et troisième caractères représentent la sous-adresse précise de destination du message : celle-ci désigne une partie seulement de l'application destinataire. Par exemple, « A1 » pour désigner la sous-partie 1 de l'application A, qui constitue la destination du message.

- Le quatrième caractère représente l'adresse de l'application émettrice du message : par exemple, 'B' pour désigner l'application B.

- Les cinquième et sixième caractères représentent la sous-adresse précise d'origine du message : celle-ci désigne une partie seulement de l'application émettrice. Par exemple, « B1 » pour désigner la sous-partie 1 de l'application B, qui constitue la source du message.

- Le septième caractère représente le statut du message (ce champ est utilisé par le CPI pour signifier qu'un problème est survenu) : '0' signifie que tout s'est bien passé, et autre chose signifie qu'il y a eu un problème. Ce champ vaut donc toujours '0' lorsque le message est une question envoyée au CPI, et ne peut valoir autre chose que dans le cas où le message est une réponse du CPI à une question dont le traitement s'est mal passé.

- Le huitième caractère représente le type du message : '0' signifie que c'est une question (qui nécessite donc une réponse de la part du CPI), '1' que c'est juste un message d'information, etc. Par défaut, tous les messages envoyés au CPI sont donc de type '0'.

- Les caractères neuf à douze représentent le nombre de messages constituant la requête. Ce point mérite une explication. Comme on l'a déjà vu, un message CPI fait au maximum 1024 caractères de long et possède un en-tête de 48 caractères, ce qui laisse 976 caractères pour le contenu véritable du message. Dans le cas où une requête est plus longue que 976 caractères, elle est donc envoyée en plusieurs messages successifs. C'est pourquoi le nombre de messages constituant la requête en cours est toujours spécifié dans l'en-tête.

- Les caractères treize à seize représentent le numéro du message dans la requête courante. Ce champ est utile dans le cas des requêtes s'étalant sur plusieurs messages (voir l'explication juste au-dessus).

- Les caractères dix-sept à vingt représentent la longueur du corps du message (comprise entre « 0000 » et « 0976 »).

- Les caractères vingt et un à vingt-quatre représentent un champ réservé (ce champ est pour l'instant inutilisé et doit prendre la valeur « 0000 »).

- Enfin, les caractères vingt-cinq à quarante huit représentent la zone « return data » (ou valeur de retour). C'est une zone dont la valeur est retournée telle quelle dans le message contenant la réponse du CPI : c'est une zone de contrôle.

Précisons tout de même que l'évolution du CPI (dans laquelle s'intègre l'API réalisée) prévoit que, dorénavant, les requêtes seront effectuées en un seul message : dans cette optique, les valeurs des champs « nombre de messages » et « numéro de message » seront pour l'instant stockées « en dur ». Chaque champ vaudra « 0001 ».

Si une application A1 veut dialoguer avec le CPI, elle n'a donc qu'à fournir les informations suivantes : la destination (sur trois caractères), la zone « return data » (facultative), et le corps du message. Tout le reste est soit commun à toutes les requêtes (la source, le statut), soit commun à toutes les applications car stocké « en dur » (le type du message, le nombre de messages par requêtes, le numéro du message, le champ réservé), soit calculé (la longueur du corps du message). La source étant commune à toutes les requêtes, elle n'a à être précisée qu'une seule fois. De même, le statut par défaut est toujours '0'.

La première étape dans la conception de cette API a été de mettre en place le formatage automatique des messages CPI à partir des informations fournies par l'utilisateur. Pour cela, on crée une classe nommée **CPIMessage** qui contient toutes les informations du message qui ne sont pas stockées « en dur ». Ainsi, les attributs de cette classe seront la source, la sous-source, la destination, la sous-destination, le statut, la zone « return data » et le corps du message. Mais cette classe définit uniquement les informations propres à chaque message, et ne fournit aucun outil pratique pour les manipuler.

C'est la classe **CPIClient** qui va se charger de la communication avec le CPI : elle fournit les méthodes de base pour envoyer et recevoir des messages. Cette classe permet tout d'abord de gérer l'établissement et la fermeture de la connexion réseau (méthodes ***connect*** et ***disconnect***), puis d'envoyer et de recevoir des informations via cette connexion réseau (***sendMsg*** et ***receiveMsg***). C'est dans ces deux méthodes que seront effectuées les opérations de conversion permettant respectivement de passer d'un objet de type CPIMessage à un message au format « transactionnel » et d'un message au format « transactionnel » à un objet de type CPIMessage. Ces deux méthodes doivent être synchronisées, car deux « threads » ne peuvent pas avoir accès à la connexion en même temps. En outre, si cette connexion est perdue, aucune requête ne pourra arriver au CPI, ce qui est une situation intolérable. D'autre part, les problèmes concernant le fonctionnement « multi-threads » de notre API commencent à apparaître. Supposons qu'un « thread » th1 envoie un message m1 sur une connexion à un temps t1 : il attend ensuite la réponse. Supposons qu'un autre « thread » th2 envoie un message m2 sur cette même connexion à un temps t2 > t1, avant que le « thread » th1 ait reçu sa réponse. Pour une raison quelconque, la réponse r2 au message m2 arrive à un temps t3 > t2, toujours avant que la réponse à m1 ne soit parvenue à th1. Alors, th1, en attente de sa réponse, recevra r2 : comme il n'existe pas à proprement parler d'identifiant pour les messages, th1 croira bel et

bien que r2 est la réponse à sa propre requête m1. De même, lorsque r1, la réponse à m1, parviendra au « thread » th2 au temps t4 > t3, celui-ci croira avoir reçu la réponse à m2. Pour régler ce problème, il existe une solution simple : il suffit de synchroniser la séquence « envoi du message + réception du message ». Ce qui revient à dire qu'un « thread » verrouille l'objet tant qu'il n'a pas reçu sa réponse. Mais la classe **CPIClient** ne devait remplir que les opérations de base vis-à-vis du CPI. Ici, ce n'est plus une fonction de base, puisqu'elle effectue, de façon synchronisée, à la fois l'envoi et la réception du message CPI. Il faut donc, soit redéfinir **CPIClient**, soit définir une classe au-dessus de CPIClient, qui l'utilise et qui se charge des opérations de haut niveau vis-à-vis du CPI et des messages (c'est-à-dire construction du message à envoyer et communication complète avec le CPI.) Cette solution semble le mieux convenir à l'API demandée, puisque l'objectif était, au final, d'obtenir une classe directement utilisable par l'utilisateur désireux de dialoguer en toute simplicité avec le CPI (ce qui exclut bien entendu qu'il ait à gérer tous les problèmes de format de message qui ne le concernent pas, ainsi que toutes les informations du message qu'il n'utilise pas.) D'autre part, **CPIClient** n'a pas à être une classe singleton (notamment parce qu'il peut exister plus d'une connexion à établir, si l'on dialogue directement avec plusieurs AACCPI ou plusieurs IACCPI), et seule la nouvelle classe, nommée **CPISession**, devra satisfaire cette contrainte.

La classe **CPISession** possède donc une méthode ***createRequest*** (surchargée trois fois), qui, à partir des informations de base fournies par le client, construit un CPIMessage prêt à être envoyé. Elle possède également une méthode ***doRequest*** qui se charge, de façon synchronisée, de l'envoi d'un message et de la réception de la réponse correspondante en provenance du CPI. Cependant, tout cela n'est pas suffisant. En effet, un autre problème a été évoqué précédemment : que se passe-t-il si la connexion utilisée (par CPIClient, et donc par CPISession) est soudain perdue ? Aucune requête ne pourra plus être acheminée… Pour pallier ce problème, il existe, dans les cas où on se connecte directement aux AACCPI ou aux IACCPI, une solution élégante : au lieu d'utiliser une seule connexion, on en utilise plusieurs (qui correspondent chacune à un IACCPI ou un AACCPI), et les requêtes sont réparties sur toutes ces connexions. Ainsi, si une connexion est perdue, les requêtes peuvent être redirigées vers une autre connexion valide, et, de surcroît, le traitement des requêtes s'en trouve accéléré. Evidemment, dans le cas où l'on parle directement au CPI, cette solution n'est pas applicable, puisqu'il ne supporte qu'une seule connexion par application. Par contre, on peut essayer de restaurer les connexions perdues, et ainsi on augmente considérablement les chances de garder au moins une connexion utilisable pour communiquer avec le CPI ! Cette solution a d'ailleurs été retenue, car il est très important de minimiser la probabilité de l'événement « plus aucun message ne peut être envoyé, car il n'existe plus aucune connexion valide ».

Cependant, cette solution n'est pas si simple à mettre en place : elle suppose d'utiliser une partie du temps accordé aux requêtes pour essayer de restaurer la ou les connexion(s) perdue(s). La meilleure solution est peut-être la suivante : dans l'instance de **CPISession**, on crée un thread indépendant dont la seule tâche est justement d'essayer en permanence de restaurer les connexions perdues. C'est au processeur que reviendra la tâche de répartir les temps d'exécution entre le programme principal et le « thread » indépendant. Dans la première version de l'API, la méthode ***run*** exécutée par le « thread » était directement implémentée dans

la classe **CPISession**. Cependant, cela posait un problème qu'il a fallu résoudre. Pour pouvoir lancer un « thread » exécutant la méthode ***run*** implémentée dans la classe **CPISession**, on est obligé :

- soit de faire de **CPISession** une classe implémentant l'interface « Runnable »,
- soit de faire hériter la classe **CPISession** de la classe **Thread**.

Dans les deux cas, la méthode ***run*** doit être déclarée comme étant une méthode publique, donc accessible à l'extérieur de la classe. En effet, la méthode ***run*** est déclarée comme publique à la fois dans l'interface « Runnable » et dans la classe **Thread**. Si **CPISession** implémente « Runnable », elle doit conserver, pour les méthodes de cette interface, les mêmes modificateurs de visibilité. De même, si **CPISession** hérite de la classe **Thread**, elle ne peut restreindre l'accès aux méthodes de sa superclasse qu'elle surcharge. Dans les deux cas, donc, la méthode ***run*** sera publique. Or, seules les méthodes destinées à être utilisées par des classes extérieures doivent normalement être déclarées comme publiques. La méthode ***run*** de la classe **CPISession**, quant à elle, n'est, ou ne devrait être, utilisée que par le « thread » dont la tâche est de restaurer les connexions perdues, créé à l'intérieur de la classe **CPISession** et en aucun cas accessible à d'autres classes. La méthode ***run*** n'a donc pas du tout le bon profil pour être une méthode publique. La solution retenue pour pallier ce problème, est la suivante : définir une classe interne privée, qui implémente l'interface « Runnable », et qui contient donc la méthode ***run***. Ainsi, la méthode ***run*** peut être déclarée publique tout en n'étant pas visible à l'extérieur de la classe **CPISession**, puisqu'elle est implémentée dans une classe qui, elle, est invisible à l'extérieur de **CPISession**.

Enfin, il restait à faire de la classe **CPISession** une classe singleton : une seule instance de cette classe doit exister à la fois. La solution à ce problème est purement technique et consiste en une astuce d'implémentation. Le principe consiste à ne définir pour cette classe qu'un constructeur par défaut privé, donc qui ne peut être appelé par une autre classe. Ainsi, aucune instance de **CPISession** ne peut être créée à l'extérieur de cette classe par appel au constructeur. Puis, on définit un attribut de classe de type **CPISession** nommé *MySession* (destiné en fait à recevoir la seule instance de cette classe), et une méthode qui crée un nouvel objet si *MySession* n'a jamais été initialisé par appel au constructeur privé, et qui renvoie *MySession* sinon. Ainsi, le seul objet **CPISession** existant est toujours *MySession* : la classe **CPISession** ne possède donc bien qu'une seule instance.

En ce qui concerne la documentation de l'API, la solution retenue a été de fournir la documentation en format html fournie par la commande « javadoc » : je ne détaillerai pas le fonctionnement de cette documentation (générée à l'aide de balises insérées dans les commentaires), mais des exemples de fichier générés par « javadoc » sont fournis en annexe 10. L'interface de la classe CPIMessage est fournie en annexe 7, celle de la classe CPISession en annexe 8, et le listing complet de la classe CPIClient en annexe 9.

Maintenant que l'API a été décrite en détails, observons son fonctionnement. Supposons que trois connexions soient disponibles (c'est-à-dire que l'utilisateur ait fourni, grâce à la méthode ***addConnection***, trois couples (hôte, port) potentiellement utilisables pour effectuer les requêtes, correspondant à trois AACCPI ou trois IACCPI). Ces connexions sont pour l'instant considérées comme non valides, puisqu'on n'a pas encore essayé de

les établir. Appelons-les C1 et C2. Supposons à présent que le « thread » chargé de l'établissement (et de la reconnexion) des connexions non valides se mette à tenter de les établir. Pendant ce temps, deux « threads » différents th1 et th2 demandent chacun à faire une requête. Si aucune connexion n'est encore établie, ils sont mis en attente. Si cette situation se prolonge trop longtemps, leurs requêtes seront abandonnées (libre à l'utilisateur de réitérer ces requêtes). Si, par contre, une connexion (par exemple C1) finit par être établie, alors th1 et th2 enverront chacun leur requête sur C1. Si les deux connexions C1 et C2 sont établies, alors th1 enverra son message sur C1 et th2 enverra le sien sur C2. Supposons alors que th1 et th2 envoient à présent des messages à intervalles réguliers. A chaque fois que l'instance de CPISession reçoit une demande de requête (par un appel à la méthode ***doRequest***), il « aiguille » le message sur la prochaine connexion valide. Par exemple, imaginons qu'un troisième « thread » th3 se mette également à faire des requêtes, et que C1 et C2 soient toutes deux valides (et qu'il n'y ait pas d'autre connexion valide). A un instant t, th1 émet une requête r1-1 qui est alors envoyée sur la connexion C1. Puis, th2 émet une requête r2-1 qui est envoyée sur la connexion C2. Ensuite, th3 émet une requête r3-1 qui est envoyée, à son tour, sur C1. A nouveau, th1 émet une requête r1-2, qui est alors envoyée sur C2, etc. Le fonctionnement se poursuit de cette façon. Voyons alors ce qui se passe si on rajoute une connexion C3 (par un appel à ***addConnection***) à la liste des connexions utilisables. Dès que le « thread » chargé d'établir les connexions a réussi à établir C3, une requête sur trois est « aiguillée » sur C3. Par exemple, si th2 est en train d'effectuer une requête à un temps t1 sur C2, et que C3 est établie au temps t2 > t1, alors si th3 effectue à son tour une requête à un temps t3 > t2, elle sera envoyée sur C3. En effet, le « thread » établissant les connexions travaille indépendamment du programme principal qui s'occupe de l'envoi et de la réception des messages, et les connexions ajoutées pendant le fonctionnement sont donc établies en « temps réel », et utilisables aussitôt.

Examinons un autre cas. Supposons que l'on ait deux « threads » th1 et th2 qui effectuent des requêtes, et trois connexions valides C1, C2 et C3. Imaginons à présent qu'un problème survienne sur l'une des connexions : supposons que th2 envoie un message m1 par la connexion C2. Le message arrive à destination, mais avant que la réponse n'ait pu être envoyée, la connexion est perdue. Que se passe-t-il alors ? La méthode ***doRequest*** appelée par th2 va signaler au « thread » chargé de restaurer les connexions perdues que la connexion C2 ne fait actuellement plus partie des connexions valides, et qu'elle doit donc être établie à nouveau. Puis, cette même méthode ***doRequest*** va récupérer la prochaine connexion valide (c'est-à-dire C3), et va réessayer d'effectuer la requête sur cette connexion. Bien entendu, entre temps, un autre message m2 aura peut-être été envoyé par un autre « thread », mais au moins m1 ne sera pas perdu. En effet, comme je l'ai déjà signalé, il n'y a qu'un seul cas où un message est perdu : s'il n'existe plus aucune connexion valide, et ce depuis un certain temps. Dans ce cas, la requête n'est jamais effectuée, mais la chose est signalée à l'utilisateur de la classe **CPISession**, sous la forme d'une exception lancée par la méthode ***doRequest***. Encore une fois, si la connexion C3 finit par être rétablie, aussitôt les messages en provenance de th1 et th2 seront automatiquement répartis sur C1, C2 et C3, grâce au mécanisme détaillé dans le paragraphe précédent. Ainsi, les risques qu'un message soit perdu sont devenus minimes, et en aucun cas un message ne peut être abandonné sans que l'utilisateur de l'API en soit averti.

On peut dès lors se demander : est-on sûr que l'exclusion mutuelle est bien réalisée ? En effet, c'était l'un des points à vérifier pour garantir le fonctionnement sécurisé de notre API avec plusieurs « threads ». On s'aperçoit que la seule situation qui pourrait provoquer un blocage de l'application est la suivante : si un « thread » reste bloqué dans la séquence « envoi d'un message + réception de la réponse ». En effet, on peut très bien imaginer qu'à un moment donné, un des IACCPI soit très lent dans son traitement des requêtes, et que le « thread » qui lui a envoyé un message attende sa réponse pendant très longtemps. Pendant ce temps-là, aucun autre « thread » ne peut entrer dans la section critique « envoi d'un message + réception de la réponse ». Mais, en réalité, cette situation ne peut pas se produire : en effet, un « Time Out » a été mis en place pour toutes les opérations de lecture / écriture dans la « socket ». Ce qui veut dire que, si au bout d'un temps spécifié par une variable nommée *Timeout* le « thread » n'a pas fini son opération de lecture ou d'écriture, l'opération est automatiquement interrompue et la requête est considérée comme ayant échoué. Ainsi, la méthode ***doRequest*** est déverrouillée et devient utilisable par un autre « thread ».

On peut faire une autre remarque intéressante concernant la conception de l'API. Telle qu'elle est actuellement, le « thread » chargé de restaurer les connexions tourne en permanence. En effet, comme on l'a déjà vu, ce « thread » est lancé dès que l'instance de **CPISession** est récupérée, et ne s'arrête qu'à la fermeture de la session. Une solution plus élégante aurait été de n'activer ce « thread » que lorsqu'il y a effectivement des connexions à rétablir, et de le désactiver quand il n'a aucune tâche à accomplir. Il aurait alors fallu concevoir et implémenter une interaction un peu plus complexe entre ce « thread » et le programme qui effectue la tâche principale. Cependant, les tests effectués sur l'API ont démontré que l'activité du « thread » ne perturbait pas l'envoi et la réception des messages CPI, car il n'est pas très « gourmand ». La solution du « thread » travaillant en continu, plus simple à mettre en place que l'autre, a donc été retenue car jugée plus rentable. Néanmoins, si de vrais problèmes de performance finissent par se poser, la prochaine version inclura peut-être une communication plus élaborée entre le module d'envoi et de réception des messages et le « thread » chargé des reconnexions.

Pour conclure, on peut remarquer qu'un danger subsiste, même s'il n'est pas grand. En effet, que se passe-t-il lorsque le « thread » chargé de restaurer les connexions tente d'en rétablir une, mais sans succès ? En fait, cela dépend. La plupart du temps, le « thread » est informé très rapidement que sa tentative a échoué. Ces cas-là ne posent aucun problème, car alors il s'occupe de la connexion suivante et essaye de la rétablir, puis il continue et passe encore à la suivante, etc. Par contre, il existe certains cas où le « thread » peut se retrouver bloqué au moment où il essaye de recréer une nouvelle « socket » : l'appel au constructeur peut bloquer indéfiniment. Ce cas est sans appel : telle qu'elle est implémentée actuellement, la classe **CPISession** ne propose aucune solution. Cependant, cela n'empêchera pas l'envoi de requêtes. En effet, même si le « thread » est bloqué, les messages pourront toujours être envoyés par un appel à la méthode ***doRequest*** : rappelons que les opérations de reconnexion et de dialogue avec le CPI sont indépendantes. Le problème qui subsistera néanmoins, si le « thread » reste effectivement bloqué indéfiniment, est que les connexions perdues ne seront plus rétablies. Le langage Java ne propose aucune solution simple pour éviter de rester bloqué lors de l'appel au constructeur de la classe **Socket**, et le moyen le plus adapté pour éviter cette situation est d'effectuer l'appel au constructeur dans un « thread »

séparé. La méthode ***join*** de la classe **Thread** permet ensuite d'arrêter le « thread » s'il n'a pas fini son travail au bout d'un temps déterminé, et ainsi d'empêcher qu'il se bloque indéfiniment. Une fonctionnalité supplémentaire à inclure dans l'API pourrait également être de définir des classes auxiliaires qui se chargeraient d'analyser le corps (et non plus l'en-tête) des messages en provenance du CPI, et de les traiter de façon systématique : il existe dix types de réponses CPI, chacune ayant un format bien spécifique et identifié par un numéro à trois chiffres situé tout au début du corps du message. C'est certainement la prochaine étape dans l'évolution des outils de dialogue avec le CPI.

VI. CONCLUSION

Le stage s'est très bien déroulé. Le fait de travailler dans une équipe jeune, dynamique et vraiment très sympathique a été un facteur motivant, malgré la durée importante du trajet quotidien.

Ces trois mois ont été enrichissants, notamment parce qu'ils m'ont permis d'observer le fonctionnement d'une petite équipe (10 personnes) sur toute cette période. Un autre point important est que j'ai été amené à travailler sur des projets assez variés, et ai donc acquis de nouvelles compétences. J'ai, par exemple, appris à utiliser le Javascript et les cgi. En outre, l'un de mes objectifs pour ce stage était d'acquérir une plus grande maîtrise des langages objets, et le fait de travailler sur un projet en C++ puis sur un projet en Java a largement contribué à le remplir. L'apport principal, à ce niveau, a surtout été de pratiquer le C++, que je n'avais quasiment pas utilisé durant l'année, et également de découvrir l'environnement de travail Visual C++ sous Windows NT.

Concernant le travail réalisé, je pense que les objectifs ont été atteints.

Tout d'abord, l'adaptation du projet AMS a été menée à bien. Les données sont à présent récupérées dans une base, et les tests pratiqués ont été concluants.

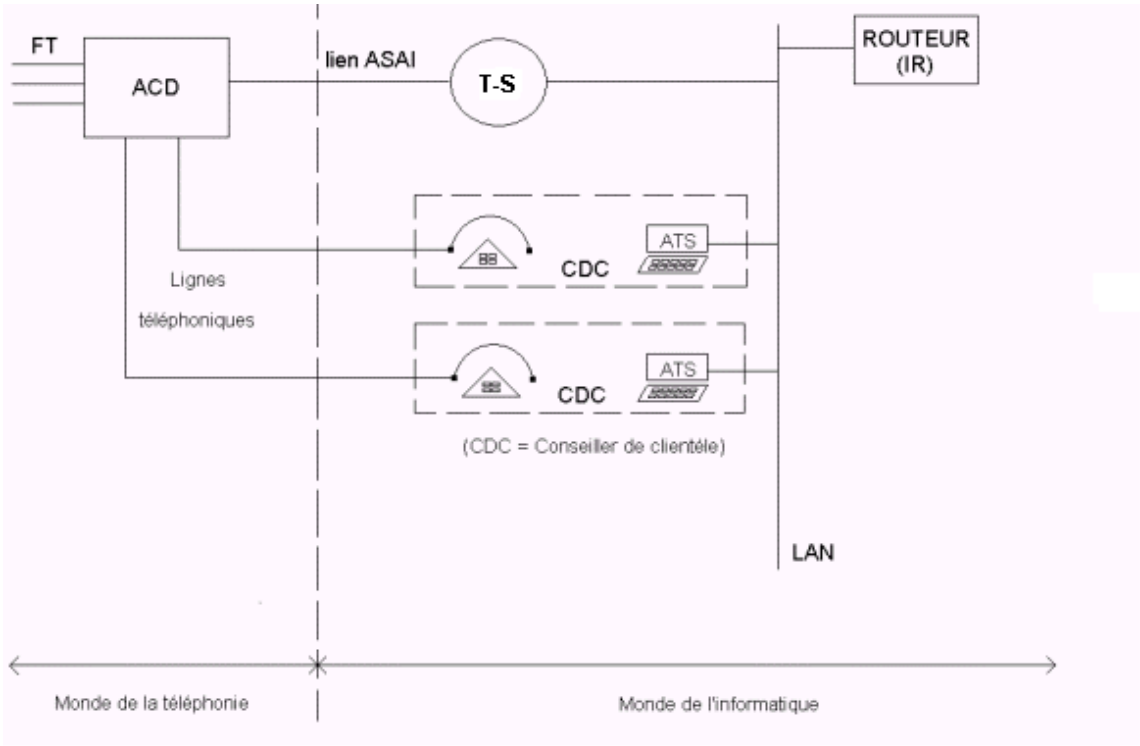
Ensuite, l'API Java, destinée à faciliter la communication avec le CPI, a également été achevée dans les temps, même si elle a connue deux versions. Après avoir conçu et défini les classes, je les ai implémentées, en réalisant des tests tout au long du développement. Tous, sauf le dernier, ont été réalisés avec une application qui simulait le fonctionnement du CPI, et le test final, qui a consisté à communiquer directement avec le CPI, s'est également bien passé.

Enfin, la maquette de l'IHM du projet URM2 a, semble-t-il, également été concluante, et a finalement été présentée aux équipes de Bouygues Telecom. Par contre, j'ai envie d'émettre ici un regret : j'aurais aimé contribuer davantage à ce projet et à l'IHM. Mon travail sur URM2 a été relativement court, et donc de ce fait limité.

Je conclurai en disant que ce stage a été, à mon sens, très positif, et que je conseille donc aux élèves de l'IIE de juger par eux-mêmes en effectuant un stage chez Datapoint.

ANNEXES

A. ANNEXE 1 : architecture du centre d'appels de Bouygues Telecom



B. ANNEXE 2 : listing de la classe SGBDConfig

```
// SGBDConfig.cpp : Implementation of CSGBDConfig

#include "stdafx.h"
#include "Sgbd.h"
#include "SGBDConfig.h"
#include "cstrtools.h"
#include "Test.h"
#define prefixe (string("conf_"))

STDMETHODIMP CSGBDConfig::Init(long addr_table)
{
		AFX_MANAGE_STATE(AfxGetStaticModuleState())

		CData *table;
		vector<string> tableTmp;
		table = (CData *) addr_table;

		tableTmp = table->GetKeys();
		if(table)
		{
			vector<CData> tableRows;
			tableRows  = table->GetVectorCData();

			for( int i=0; i<tableRows.size(); i++)
			{
				CData row = tableRows[i];
				m_CDatastock.Add( row.GetStr("CONFIGKEY").c_str(),
							row.GetStr("CONFIGVALUE").c_str());

			}

		}
		return S_OK;
		}


STDMETHODIMP CSGBDConfig::get_ProfileString(BSTR section, BSTR key, BSTR *pVal)
{

		AFX_MANAGE_STATE(AfxGetStaticModuleState())

		string str,res;
		int i;
		bool ok=false;
		vector<string> sections;
		vector<string> keys;
		string section_name,key_name;
		string error1, error2;
		CComBSTR bstrTemporaire;
		error1 = "key_error";
		error2 = "section_error";
		str = m_CDatastock.GetStr(prefixe+"sections");

		// str = la valeur associée à la clé "sections", c'est-à-dire "mail, gui, com, custom, tserver, engine,
		// OgmTabDlg"

		USES_CONVERSION;

		section_name = OLE2A(section);
		key_name = OLE2A(key);
```

```
sections = CStrTools::Tokenize(str, ",", false, " ");                    // sections = |"mail"|"gui"|"com"|...|
                                                                         // niveau 2 de  l'arbre

for (i=0; i<sections.size() && !ok; i++)  // pour "mail", "gui", "com", "custom", "tserver", "engine",
                                          // "OgmTabDlg"
{
        if (sections[i]==section_name)          ok=true;
}


if (ok)
{
        str = m_CDatastock.GetStr(prefixe+sections[i-1]);   // renvoie la valeur associée à la clé
                                                            // sections[i-1] (= associée à "mail" ou
                                                            // "gui" ou etc.)

        keys = CStrTools::Tokenize(str, ",", false, " ");       // niveau 3 de l'arbre
        ok=false;

        for (i=0; i<keys.size() && !ok; i++)
        {
                if (keys[i]==key_name)          ok=true;
        }
        if (ok)
        {
                res=m_CDatastock.GetStr(prefixe+keys[i-1]);      // feuille de l'arbre correspondant
à                                                                // la valeur à  retourner, càd celle
                                                                 // associée à la section et à la key

                bstrTemporaire = A2OLE(res.c_str());
                *pVal = bstrTemporaire.Copy();
                return S_OK;
        }
        else
        {
                bstrTemporaire = A2OLE(error1.c_str());  // la clé n'est pas valable
                *pVal = bstrTemporaire.Copy();
                return S_OK;
        }
}
else
{
        bstrTemporaire = A2OLE(error2.c_str());  // la section n'est pas valable
        *pVal = bstrTemporaire.Copy();
        return S_OK;
}

}
```

C. ANNEXE 3 : extrait du listing de la classe CWndParserDb

```
// WndParserDb.cpp: implementation of the CWndParserDb class.
//
//////////////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "AtsLib.h"
#include "WndParserDb.h"
#include "cstrtools.h"
#include "WndParser.h"
#include "log.h"
#define prefixe (string("conf_"))
#define prefixe2 (string("wnd_"))


extern Clog*  WndLog;

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CWndParserDb::CWndParserDb()
{
		memset(&m_dlgBoxDesc, 0, sizeof(DIALOGBOX_DESC));
}

CWndParserDb::~CWndParserDb()
{
		m_listCtrl.clear();
}

//////////////////////////////////////////////////////////////////////
// Initialize the m_dlgBoxDesc and m_wndID member variables.

BOOL CWndParserDb::ParseRootNode(CData *CDataDb)
 {

		// On recupere ici tous les paramètres associés à une window

		CString c;
		int pos;
		vector<string> tmp0, tmp, tmp2;
		int x, y, cx, cy;
		int j;
		string str;
		CString strError(_T("CWndParser::ParseRootNodeDb() failed: missing information in the database.\n"));
		m_wndName = window_courante.c_str();
	try
	{
			str = CDataDb->GetStr(prefixe2+window_courante);
			*WndLog << "windowData(" << window_courante <<")=" << str << "\n";
			tmp0 = CStrTools::Tokenize(str, "\\", false, " ");
			if (tmp0.size() != 0)			// si str n'est pas une suite d'espaces vides
			{
				for (j=0; j<tmp0.size(); j++)
				{
					tmp = CStrTools::Tokenize(tmp0[j], "->", false, " ");
					c = tmp[0].c_str();
					*WndLog << "current=" << tmp[0] << "\n";

					if (tmp[0]=="Dim")
				{
					if (tmp.size()>1)
```

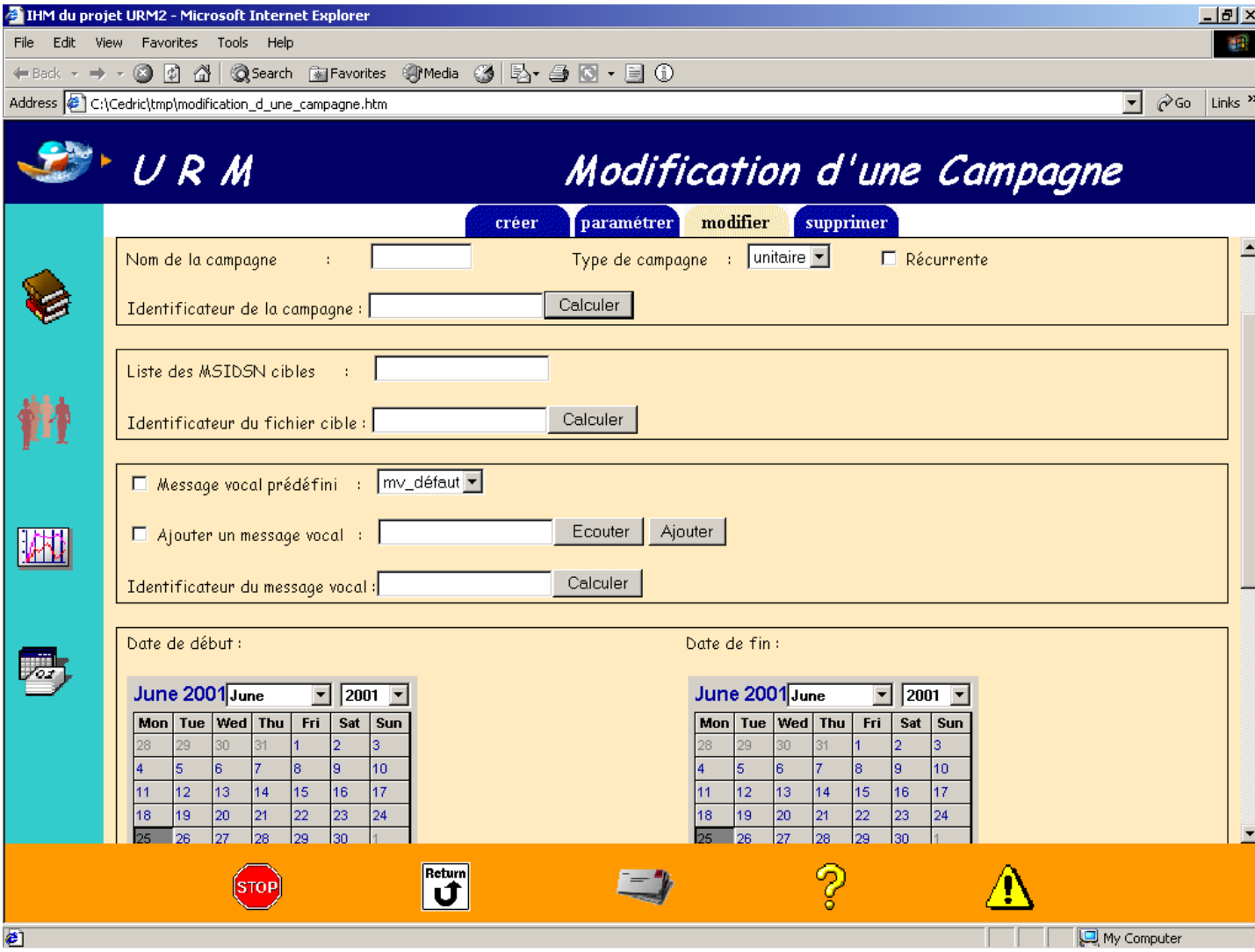
```
{
        tmp2 = CStrTools::Tokenize(tmp[1], ",", false, " ");
        if (tmp2.size()>3)
        {
                x = ReadInteger((char*)tmp2[0].c_str());
                y = ReadInteger((char*)tmp2[1].c_str());
                cx = ReadInteger((char*)tmp2[2].c_str());
                cy = ReadInteger((char*)tmp2[3].c_str());
                m_dlgBoxDesc.rect.SetRect(x, y, x + cx, y + cy);
        }
        else throw strError;
}
else throw strError;
```

D. ANNEXE 4 : extrait du listing de la classe CTEventsWndParserDb

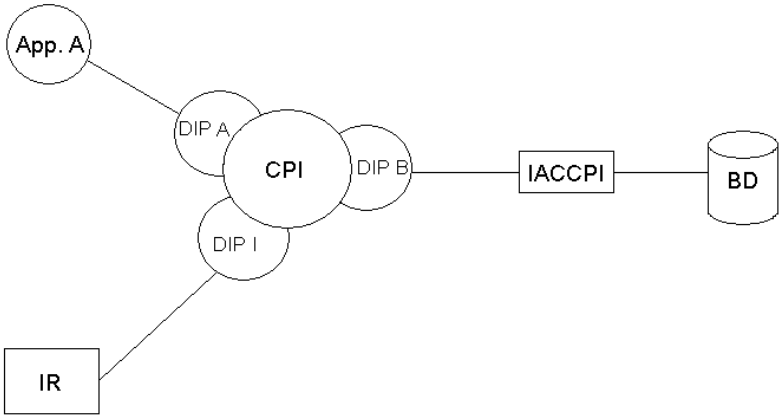
```
// TEventsWndParserDb.cpp: implementation of the CTEventsWndParserDb class.
//
//////////////////////////////////////////////////////////////////////

#include "TEventsWndParserDb.h"
#include "TEventParser.h"
#include "cstrtools.h"
#include "log.h"

#define prefixe (string("conf_"))
#define prefixe2 (string("wnd_"))

extern Clog* WndLog;


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CTEventsWndParserDb::~CTEventsWndParserDb()
{
	m_listAct.RemoveAll();
}


bool CTEventsWndParserDb::ParseActionsNode(CData *CDataDb)
{
	// On recupere ici toutes les actions associées a un profile

	CAtsAction action;
	CString c;
	CString strError(_T("ParseActionsNodeDb() failed: missing information in the database.\n"));
	int i,j,k;
	int pos;
	int nLength;
	bool ok=false,ok2=false;
	string str,str2;
	string Action;
	vector<string> tmp, tmp2, tmp3,tmp4,tmp5;
	Action = "Action";

	try
	{
		str = CDataDb->GetStr(prefixe2+profile_courant);
		tmp = CStrTools::Tokenize(str, ",", false, " ");		// tmp = |"Conditions1"|"Actions1"|
		if (tmp.size() != 0)		// si str n'est pas une suite d'espaces vides
		{
			if (tmp.size() == 2)//if2
			{
				for (i=0;i<2 && !ok;i++)
				{
					c = tmp[i].c_str();
					if ((pos = c.Find("Action")) == 0) ok=true;
				}
				if (ok)
				{
					str2 = CDataDb->GetStr(prefixe2+tmp[i-1]);
					tmp2 = CStrTools::Tokenize(str2, "\\", false, " ");
							for (j=0;j<tmp2.size();j++)
					{
						c = tmp2[j].c_str();
						if ((pos = c.Find("Hide")) == 0)
```

```
{
        tmp3 = CStrTools::Tokenize(tmp2[j], " ",
                                        false, " ");
        if (tmp3.size()>1)
        {
                action.m_Name = tmp3[1].c_str();
                action.m_dwType =
                        ActionHideWindow;
        }
        else throw strError;
}
```

E. ANNEXE 5 : maquette de l'IHM du projet URM2



F. ANNEXE 6 : schéma du fonctionnement du CPI



G. ANNEXE 7 : interface de la classe CPIMessage

```
package datapoint.cpi;
import java.util.*;

/**
*    Classe définissant le format d'un message CPI.
*
*    @author DATAPOINT, 2001
*    @version 1.2
*/

public class CPIMessage {

  private static int CPI_DEST_SIZE = 1;
  private static int CPI_SUBDEST_SIZE = 2;
  private static int CPI_SOURCE_SIZE = 1;
  private static int CPI_SUBSOURCE_SIZE = 2;
  private static int CPI_STATUS_SIZE = 1;
  private static int CPI_MSGTYPE_SIZE = 1;
  private static int CPI_MSGNBR_SIZE = 4;
  private static int CPI_MSGNUM_SIZE = 4;
  private static int CPI_DTLGTH_SIZE = 4;
  private static int CPI_RESERVED_SIZE = 4;
  private static int CPI_RETURNDATA_SIZE = 24;
  private static int CPI_DEST_POS = 1;
  private static int CPI_SUBDEST_POS = CPI_DEST_POS+CPI_DEST_SIZE;
  private static int CPI_SOURCE_POS = CPI_SUBDEST_POS+CPI_SUBDEST_SIZE;
  private static int CPI_SUBSOURCE_POS = CPI_SOURCE_POS+CPI_SOURCE_SIZE;
  private static int CPI_STATUS_POS = CPI_SUBSOURCE_POS+CPI_SUBSOURCE_SIZE;
  private static int CPI_MSGTYPE_POS = CPI_STATUS_POS+CPI_STATUS_SIZE;
  private static int CPI_MSGNBR_POS = CPI_MSGTYPE_POS+CPI_MSGTYPE_SIZE;
  private static int CPI_MSGNUM_POS = CPI_MSGNBR_POS+CPI_MSGNBR_SIZE;
  private static int CPI_DTLGTH_POS = CPI_MSGNUM_POS+CPI_MSGNUM_SIZE;
  private static int CPI_RESERVED_POS = CPI_DTLGTH_POS+CPI_DTLGTH_SIZE;
  private static int CPI_RETURNDATA_POS = CPI_RESERVED_POS+CPI_RESERVED_SIZE;
  private static int CPI_HEADER_SIZE = 48;
  private static int CPI_MSG_MAX_SIZE = 1024;
  private static String CPI_MSGNBR_VAL = "0001";
  private static String CPI_MSGNUM_VAL = "0001";
  private static String CPI_MSGTYPE_VAL = "0";
  private static String CPI_RESERVED_VAL = "0000";

  private char   dest;
  private String subDest;
  private char   source;
  private String subSource;
  private char   status;
  private String returnData;
  private String msgBody;

  public CPIMessage() {}
  public void setDest(char dest) {}
  public void setSubDest(String subDest) {}
  public void setSource(char source) {}
  public void setSubSource(String subSource) {}
  public void setStatus(char status) {}
  public void setReturnData(String returnData) {}
  public void setMsgBody(String msgBody) {}
  public char getDest() {}
  public String getSubDest() {}
  public char getSource() {}
  public String getSubSource() {}
```

public char getStatus() {}
public String getReturnData() {}
public String getMsgBody() {}

static int getCPI_DEST_SIZE() {}
static int getCPI_SUBDEST_SIZE() {}
static int getCPI_SOURCE_SIZE() {}
static int getCPI_SUBSOURCE_SIZE() {}
static int getCPI_STATUS_SIZE() {}
static int getCPI_MSGTYPE_SIZE() {}
static int getCPI_MSGNBR_SIZE() {}
static int getCPI_MSGNUM_SIZE() {}
static int getCPI_DTLGTH_SIZE() {}
static int getCPI_RESERVED_SIZE() {}
static int getCPI_RETURNDATA_SIZE() {}
static int getCPI_DEST_POS() {}
static int getCPI_SUBDEST_POS() {}
static int getCPI_SOURCE_POS() {}
static int getCPI_SUBSOURCE_POS() {}
static int getCPI_STATUS_POS() {}
static int getCPI_MSGTYPE_POS() {}
static int getCPI_MSGNBR_POS() {}
static int getCPI_MSGNUM_POS() {}
static int getCPI_DTLGTH_POS() {}
static int getCPI_RESERVED_POS() {}
static int getCPI_RETURNDATA_POS() {}
static int getCPI_HEADER_SIZE() {}
static int getCPI_MSG_MAX_SIZE() {}
static String getCPI_MSGNBR_VAL() {}
static String getCPI_MSGNUM_VAL() {}
static String getCPI_MSGTYPE_VAL() {}
static String getCPI_RESERVED_VAL() {}
public String toString() {}

H. ANNEXE 8 : interface de la classe CPISession

```
package datapoint.cpi;
import java.util.*;
import java.net.*;
import java.io.*;

/**
*    Classe singleton (une seule instance peut exister à la fois) permettant l'envoi et la réception de messages CPI par
*    un mécanisme "thread-safe".
*
*    @author DATAPOINT, 2001
*    @version 1.2
*/
public class CPISession {

    private static CPISession MySession;
    private char source;
    private String subSource;
    private Vector nc_hosts;
    private Vector nc_ports;
    private Vector c_hosts;
    private Vector c_ports;
    private Vector nc_CPIClients;
    private Vector c_CPIClients;
    private int indice_courant;
    private int timeOut;
    private boolean fin;
    private boolean debug ;

//  Constructeur par défaut (private)
    private CPISession() {}

//  Méthode synchronisée permettant de récupérer l'unique instance de CPISession.
//  @return l'unique instance de la classe CPISession.
    public synchronized static CPISession getInstance() {}

//  Méthode privée qui lance le thread chargé de restaurer les connexions
    private void Initialize() {}

//  Accesseur en écriture du Timeout, utilisé lors des lectures/écritures dans la socket. Par défaut, la valeur du
//  Timeout est de 10 secondes.
//  @param timeOut la nouvelle valeur du Timeout.
    public synchronized void setTimeOut(int timeOut) {}

//  Accesseur en écriture de la variable debug : si debug vaut true, les traces de l'application seront visibles. Dans
le //cas contraire, ces informations seront masquées. Par défaut, debug vaut true.
//  @param debug la nouvelle valeur de debug.
    public synchronized void setDebug(boolean debug) {}

//  Accesseur en écriture de l'adresse locale, utilisée lors de l'envoi des messages. Par défaut, leurs valeurs sont
//  respectivement un caractère blanc et une chaîne formée de deux caractères blancs.
//  @param source la nouvelle source.
//  @param subSource la nouvelle sous-source.
    public synchronized void setLocalAddress(char source, String subSource) {}

//  Méthode créant un CPIMessage à partir des paramètres passés en arguments. Le champ status prend la valeur
par
//  défaut '0'.
//  @param dest la destination.
//  @param subDest la sous-destination.
//  @param returnData la valeur de la zone ReturnData.
//  @param msgBody le contenu du message.
//  @return le message créé.
```

```
        public CPIMessage createRequest(char dest, String subDest, String returnData, String msgBody) {}
        public CPIMessage createRequest(char dest, String subDest, String returnData) {}
        public CPIMessage createRequest(char dest, String subDest) {}

// Méthode synchronisée permettant d'ajouter une connexion, c'est-à-dire un couple (hôte, port), à la liste de celles
// déjà utilisées pour effectuer les requêtes.
// @param host le nom de l'hôte.
// @param port le numéro du port.
        public synchronized void addConnection(String host, int port) {}

// Méthode permettant d'effectuer une requête, c'est-à-dire d'envoyer une question et de récupérer la réponse
// correspondante.
// @param message le message contenant la question à envoyer.
// @return le CPIMessage contenant la réponse à la question envoyée.
// @throws Exception si aucune connexion n'a pu être obtenue pour exécuter la requête.
        public CPIMessage doRequest(CPIMessage message)
        throws Exception {}

// Méthode synchronisée permettant d'exécuter une requête
// @param c le CPIClient utilisé pour exécuter la requête
// @param m le CPIMessage contenant la question à envoyer
// @return le CPIMessage contenant la réponse à la question posée
        private synchronized CPIMessage doRequestSynchronized(CPIClient c, CPIMessage m)
        throws Exception {}

// Méthode permettant d'obtenir une connexion valide
// @return un CPIClient connecté
        private synchronized CPIClient getConnection() {}

// Méthode synchronisée effectuant la fermeture de la session.
        public synchronized void close() {}

// Méthode synchronisée qui renvoie la prochaine connexion à rétablir
// @param rang la position de la connexion précédemment en cours de traitement
// @param incremente la distance entre les positions de l'ancienne et de la nouvelle connexion
// @return la position de la prochaine connexion
        private synchronized int doTest(int rang, int incremente) {}

// Méthode synchronisée ajoutant une connexion à l'ensemble des connexions valides
// @param o1 le CPIClient associé à la connexion
// @param o2 le nom de l'hôte associé à la connexion
// @param o3 le numéro du port associé à la connexion
        private synchronized void addToConnected(CPIClient o1, String o2, Integer o3) {}

// Méthode synchronisée retirant une connexion à l'ensemble des connexions non valides
// @param indice l'indice de l'élément à retirer
        private synchronized void removeFromNonConnectedAt(int indice) {}

// Méthode synchronisée ajoutant une connexion à l'ensemble des connexions non valides
// @param o1 le CPIClient associé à la connexion
// @param o2 le nom de l'hôte associé à la connexion
// @param o3 le numéro du port associé à la connexion
        private synchronized void addToNonConnected(CPIClient o1, String o2, Integer o3) {}

// Méthode synchronisée retirant une connexion à l'ensemble des connexions valides
// @param indice l'indice de l'élément à retirer
        private synchronized void removeFromConnectedAt(int indice) {}

// Classe interne privée implémentant Runnable et contenant la méthode run utilisée par le thread chargé de restaurer
les //connexions
        private class ConnectThread implements Runnable {

                public ConnectThread() {}

                public void run() { }
```

```
        }

}
```

I. ANNEXE 9 : listing de la classe CPIClient

```
package datapoint.cpi;
import java.util.*;
import java.net.*;
import java.io.*;

/**
*     Classe implémentant un client CPI.
*
*     @author DATAPOINT, 2001
*     @version 1.2
*/
public class CPIClient {

//     Attribut maSocket contenant la socket ouverte lors de la connexion
//     @see CPIClient # boolean connect(String,int)
//     @see CPIClient # boolean connect(String,int,int)
  private Socket maSocket;

/**
*     Constructeur par défaut de CPIClient.
*/
  public CPIClient() {
  }

/**
*     Méthode synchronisée qui gère la connexion du CPIClient au serveur.
*     @param host le nom de l'hôte à atteindre.
*     @param port le numéro du port utilisé.
*     @return true si tout s'est bien passé.
*     @throws Exception si la connexion a échoué.
*/
  public synchronized boolean connect(String host, int port)
  throws Exception
  {
      try
      {
                maSocket = new Socket(host,port);
                return true;
      }
      catch (Exception e)
      {
                throw e;
      }
  }

/**
*     Méthode synchronisée qui gère la connexion du CPIClient au serveur, en spécifiant un Timeout pour les
opérations *   de lecture/écriture dans la socket.
*     @param host le nom de l'hôte à atteindre.
*     @param port le numéro du port utilisé.
*     @param timeOut la durée autorisée avant le Timeout.
*     @return true si tout s'est bien passé.
*     @throws Exception si la connexion a échoué.
*/
  public synchronized boolean connect(String host, int port, int timeOut)
```

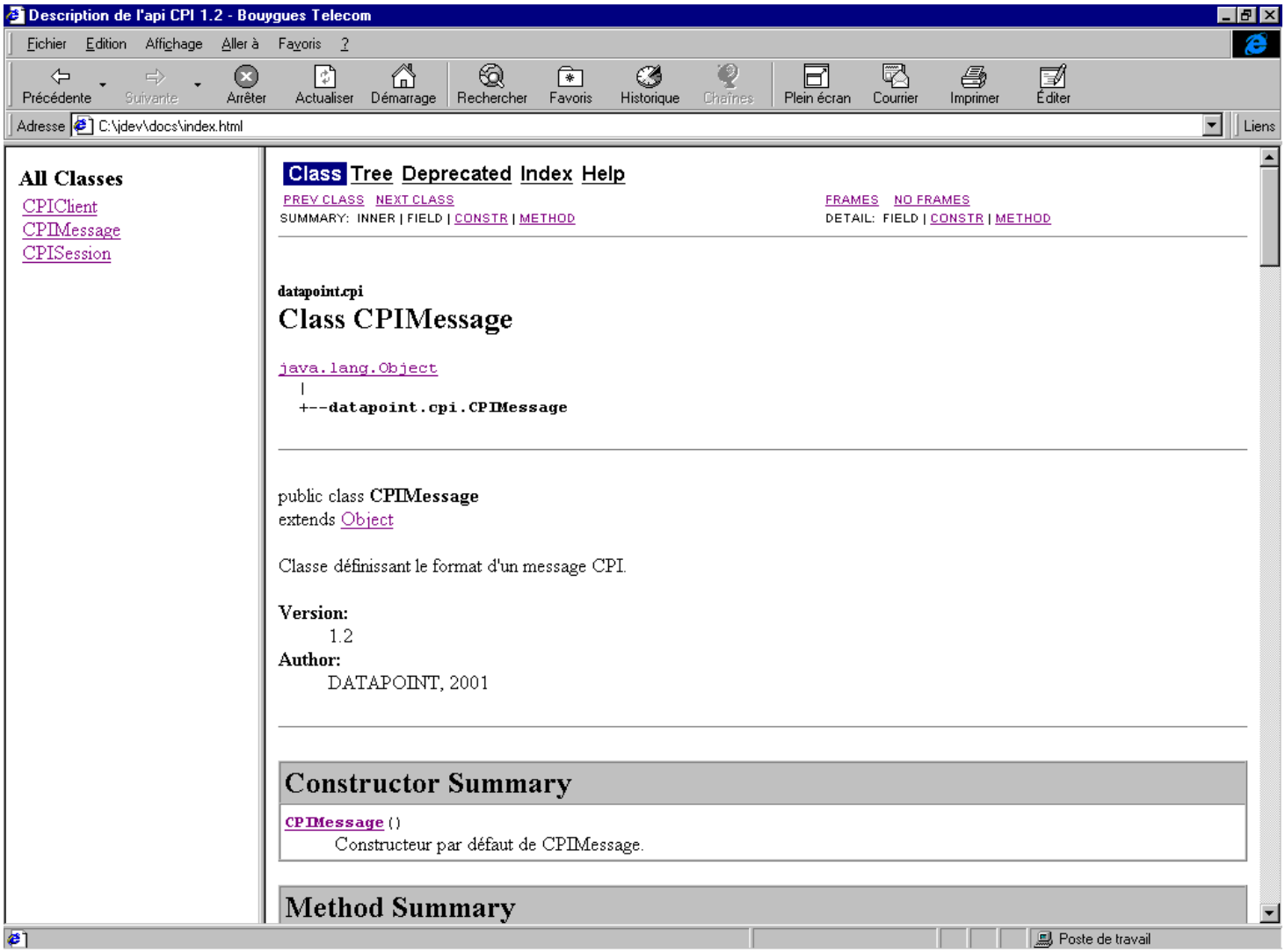
```
 throws Exception
 {
    try
    {
            maSocket = new Socket(host,port);
            maSocket.setSoTimeout(timeOut);
            return true;
    }
    catch (Exception e)
    {
            throw e;
    }
 }

/**
*    Méthode synchronisée qui gère la fermeture de la socket.
*/
    public synchronized void disconnect() {
    try
    {
            maSocket.close();
    }
    catch (IOException e)
    {
    }
 }

/**
*    Méthode synchronisée déclenchant l'envoi d'un CPIMessage au serveur.
*    @param message le message à envoyer.
*    @throws IOException si un problème est survenu lors de l'écriture dans la socket.
*/
 public synchronized void sendMsg(CPIMessage message)
    throws IOException
    {
    String str = message.toString();
    try
    {
            PrintWriter ChaineMessage = new PrintWriter (maSocket.getOutputStream(),true);
            ChaineMessage.print(str);
            ChaineMessage.flush();
    }
    catch (IOException e)
    {
            throw e;
    }
 }

/**
*    Méthode permettant la lecture complète d'un flux en provenance du serveur par le CPIClient.
*    @param in le flux d'entrée.
*    @param alire le nombre de caractères à lire.
*    @param buffer le tableau où sont stockés les caractères lus.
*/
 private void readInputStream(InputStream in, int alire, byte[] buffer)
 throws Exception
 {
    int lu=0,totallu=0;
    try
    {
            while (totallu != alire)
            {
                    lu=in.read(buffer,0,alire-totallu);
                    if (lu==-1)
                            throw new SocketException("Connection reset by peer");
```

```
                totallu+=lu;
            }
        }
        catch (Exception e)
        {
            throw e;
        }
    }



/**
*   Méthode synchronisée déclenchant la réception par le CPIClient d'un CPIMessage en provenance du serveur.
*   @return le message reçu.
*   @throws Exception si un problème est survenu lors de la lecture dans la socket.
*/
    public synchronized CPIMessage receiveMsg()
        throws Exception
        {
        CPIMessage message = new CPIMessage();
        String line,line2;
        String tmp;
        int lgth;
        byte[] header,body;

        try
        {
            InputStream in = maSocket.getInputStream();
            header = new byte[CPIMessage.getCPI_HEADER_SIZE()];
            readInputStream(in,CPIMessage.getCPI_HEADER_SIZE(),header);
            line = new String(header);
                tmp = line.substring(CPIMessage.getCPI_DEST_POS()-1,CPIMessage.getCPI_DEST_POS()
              +CPIMessage.getCPI_DEST_SIZE()-1);
            message.setDest((tmp.toCharArray())[0]);
                tmp =
              line.substring(CPIMessage.getCPI_SUBDEST_POS()-1,CPIMessage.getCPI_SUBDEST_POS()
              +CPIMessage.getCPI_SUBDEST_SIZE()-1);
                message.setSubDest(tmp);
                tmp =
              line.substring(CPIMessage.getCPI_SOURCE_POS()-1,CPIMessage.getCPI_SOURCE_POS()
              +CPIMessage.getCPI_SOURCE_SIZE()-1);
                message.setSource((tmp.toCharArray())[0]);
                tmp =
              line.substring(CPIMessage.getCPI_SUBSOURCE_POS()-1,CPIMessage.getCPI_SUBSOURCE_P
              OS()+CPIMessage.getCPI_SUBSOURCE_SIZE()-1);
                message.setSubSource(tmp);
                tmp =
              line.substring(CPIMessage.getCPI_STATUS_POS()-1,CPIMessage.getCPI_STATUS_POS()
              +CPIMessage.getCPI_STATUS_SIZE()-1);
                message.setStatus((tmp.toCharArray())[0]);
                tmp =
              line.substring(CPIMessage.getCPI_RETURNDATA_POS()-1,CPIMessage.getCPI_RETURNDAT
              A_POS()+CPIMessage.getCPI_RETURNDATA_SIZE()-1);
                message.setReturnData(tmp);
                lgth =
              Integer.valueOf(line.substring(CPIMessage.getCPI_DTLGTH_POS()-1,CPIMessage.getCPI_DTL
              GTH_POS()+CPIMessage.getCPI_DTLGTH_SIZE()-1)).intValue();
            body = new byte[lgth];
            readInputStream(in,lgth,body);
            line2 = new String(body);
            message.setMsgBody(line2);
            return message;
        }
        catch (Exception e)
        {
```

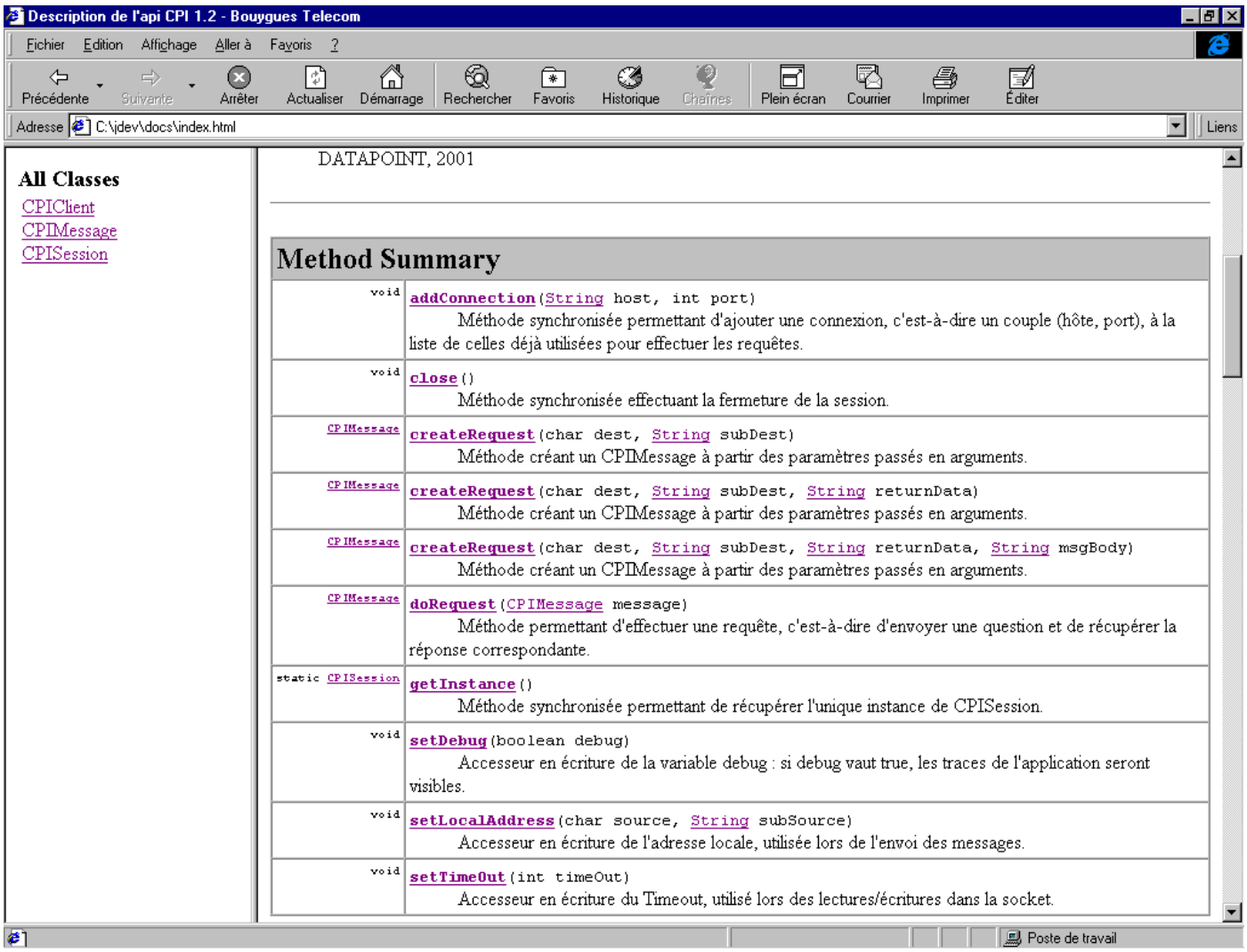
```
                throw e;
            }
        }
    }
```

J. ANNEXE 10 : Extraits de javadoc

Javadoc de CPIMessage :



Javadoc de CPISession :



Javadoc de CPIClient :