INSA Rennes – Département INFORMATIQUE

Système Expert pour Smartphones

Rapport de Conception

Romain Boillon, Olivier Corridor, Quentin Decré, Vincent Le Biannic, Germain Lemasson, Nicolas Renaud, Fanny Tollec

2010-2011

Projet sous la direction de Laurence Rozé

TABLE DES MATIERES

1. Introduction

Notre projet, intitulé « Manage Yourself » consiste à développer une application de diagnostic et de prévention d'erreur pour les Smartphone Android et iPhone. Comme décrit dans notre rapport de spécification, notre projet est principalement découpé en deux parties.

L'une est la partie mobile dans laquelle se trouvent le système expert en charge de la surveillance du Smartphone, le système de reporting qui génère les rapports d'erreurs et de bon fonctionnement, et le suivi utilisateur pour Android.

L'autre partie est le serveur sur lequel s'effectue l'apprentissage des règles nécessaires pour le système expert ainsi qu'une interface administrateur.

Ce rapport de modélisation a pour but de décrire l'architecture logicielle que nous allons développer pour notre projet. Les différents modules de l'ensemble du système sont présentés en détails pour décrire et expliciter son fonctionnement.

Différentes modélisations sont utilisées, à savoir les diagrammes de classes UML pour une représentation statique, et les diagrammes de séquences et d'activités pour une représentation dynamique.

Le rapport présente premièrement la modélisation générale du système. Nous spécifions ensuite la partie mobile, puis la partie serveur et enfin la partie simulateur.

2. Conception générale

2.1. Présentation Générale

Comme il a été présenté dans les précédents rapports, le système est constitué de deux parties :

- Partie mobile
- Partie serveur

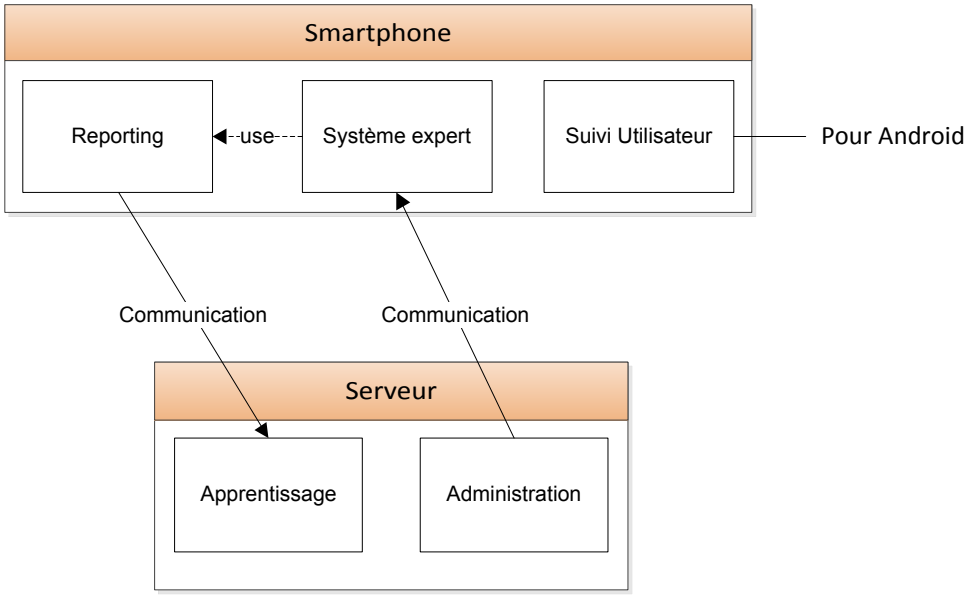

DIAGRAMME 1 REPRESENTATION DES DIFFERENTES PARTIES DU SYSTEME

Comme le montre la figure, dans chaque partie, on retrouve plusieurs composants dont le rôle est clairement défini. Si on parcourt les différents composants dans l'ordre dans lequel ils interviennent, on obtient :

- **Reporting** : maintient une base de faits observés sur le mobile et génère des rapports au format XML
- **Apprentissage** : apprend des règles à partir des rapports. Ces règles permettent de reconnaitre des situations propices aux plantages
- **Administration** : permet à l'administrateur d'associer des actions aux règles apprises. Cela donne de nouvelles règles étiquetées par des actions, qui sont envoyées sur le mobile au format XML
- **Système expert** : lit les règles, et fait de l'inférence sur la base des règles et des faits pour exécuter des actions si nécessaire.

2.2. Mobile / Serveur

Nous allons décrire ces différents modules de façon plus concrète via des diagrammes de déploiement.

2.2.1. Smartphone

On peut représenter le fonctionnement du mobile de la façon suivante :

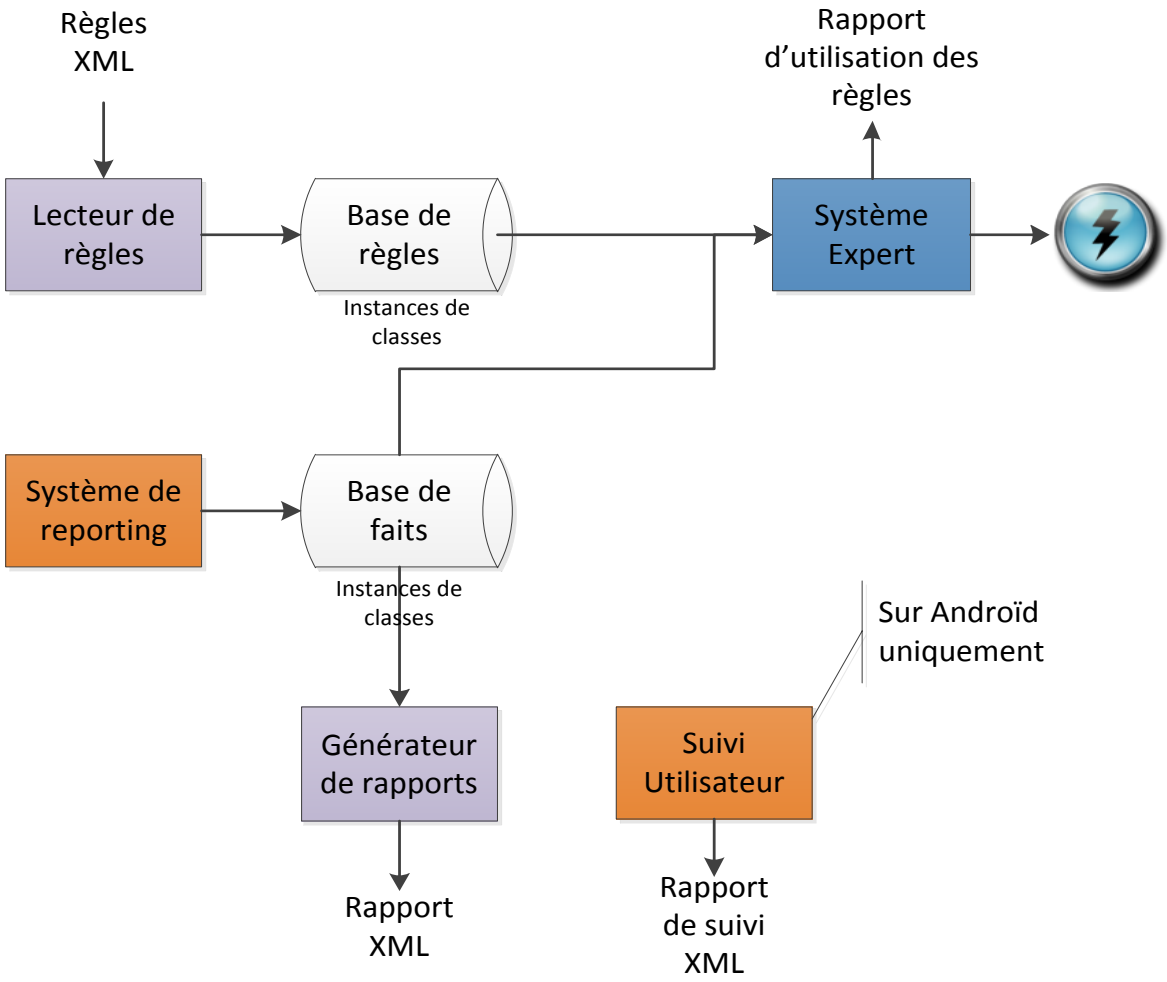

DIAGRAMME 2 - REPRESENTATION INDICATIVE DU FONCTIONNEMENT DU SYSTEME SUR LE MOBILE

En pratique, les bases de faits et de règles occupent une place prépondérante. Le système de reporting est intégré à la base de faits et le système expert est intégré à la base de reporting. Les bases prennent donc une importance capitale. Nous verrons en partie 2.3, comment les bases seront implémentées, puis, en 3.1 et 0, de quelle façon, la dynamique du système expert et du système de reporting est intégré aux bases.

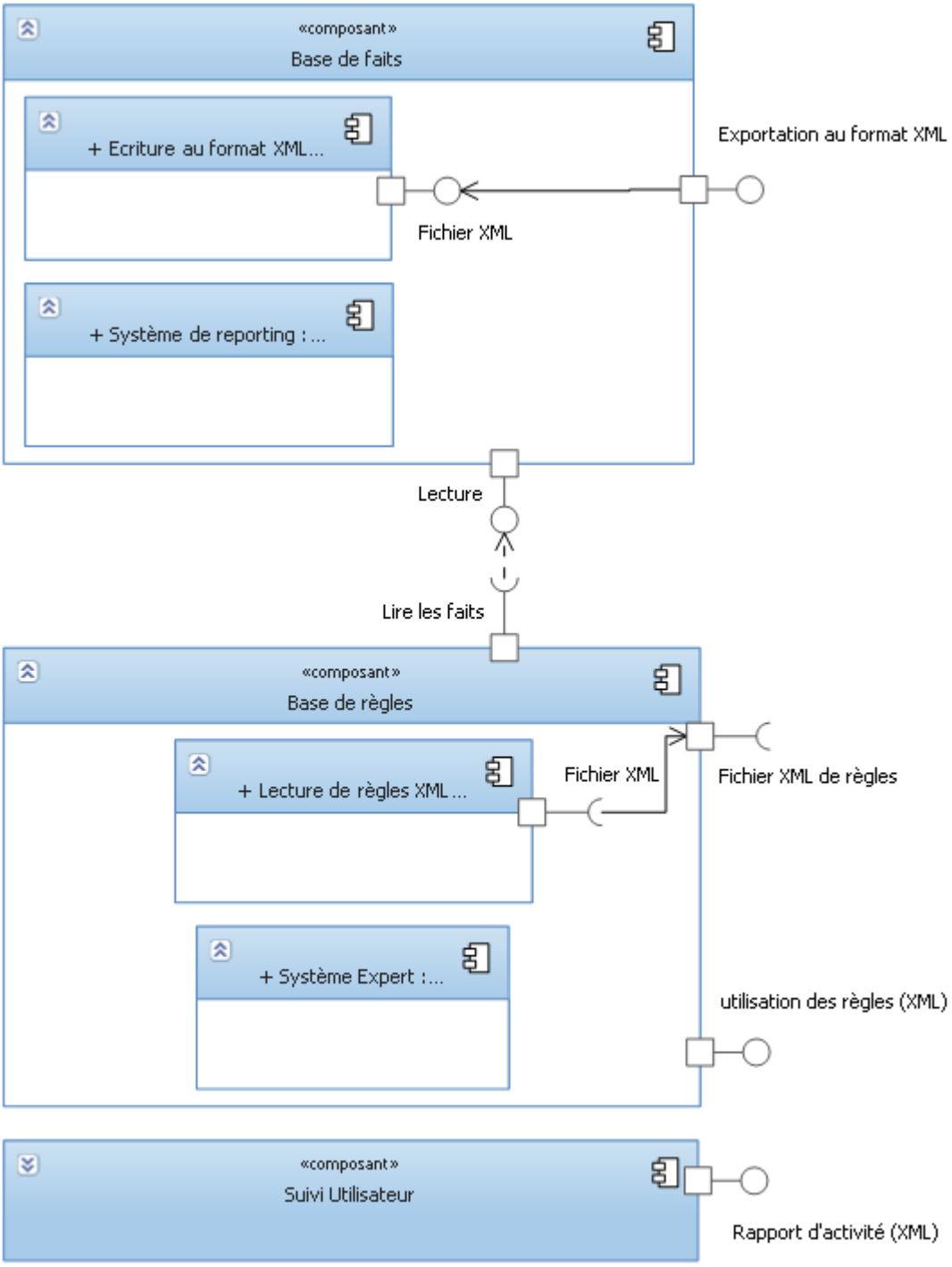

DIAGRAMME 3 - COMPOSANTS SUR LE SMARTPHONE

On retrouve ainsi sur le digramme de composants précédent, les 3 principales parties du mobile :

- **La base de faits**, qui surveille l'état du Smartphone, et qui via le système de reporting, exporte les rapports au format XML.
- **La base de règles**, qui se charge de lire les règles et de déduire les actions à appliquer sur la base des faits lus sur la base de faits. En outre, la base de règles propose un rapport XML sur l'utilisation des règles.
- **Le suivi utilisateur**

2.2.2. Serveur

Si on s'intéresse maintenant au serveur, on retrouve les entrées et sorties indiquées dans le rapport de spécification :

- Rapports XML :
  - Rapports de bon fonctionnement Androïd
  - Rapports de mauvais fonctionnement Androïd
  - Rapports de bon fonctionnement iPhone
- Fichiers de bug iPhone : cela permet de compenser l'impossibilité de générer des rapports de mauvais fonctionnement directement depuis l'iPhone. On récupère donc un rapport de bug généré par le système iOS (et non notre système de reporting), et récupéré par le serveur lors d'une synchronisation. Nous verrons comment ces fichiers sont utilisés pour produire les rapports de mauvais fonctionnement iPhone.
- Fichier XML de règles : le serveur, sur la base de ces deux premières entrées et de l'intervention d'un administrateur, peut produire un fichier XML de règles.

Sur le schéma en page suivante, on choisit dans un premier temps d'abstraire le fonctionnement interne du créateur de fichier ARFF, qui permet de produire un fichier d'apprentissage pour WEKA sur la base des rapports XML et des fichiers de bug iPhone. Nous y reviendrons plus tard.

Une fois le fichier ARFF créé, l'apprentissage se charge de rendre un résultat dans un format brut qui sera analysé, afin de communiquer les prémisses des règles à l'interface d'administration.

L'administrateur pourra alors :

- Modifier des règles existantes
- Supprimer des règles existantes

- Ajouter une ou plusieurs action(s) aux prémices de règles apprises, et les ajouter à la base de règles.

La base de règle est alors exportée au format XML, et sera envoyée sur le mobile.

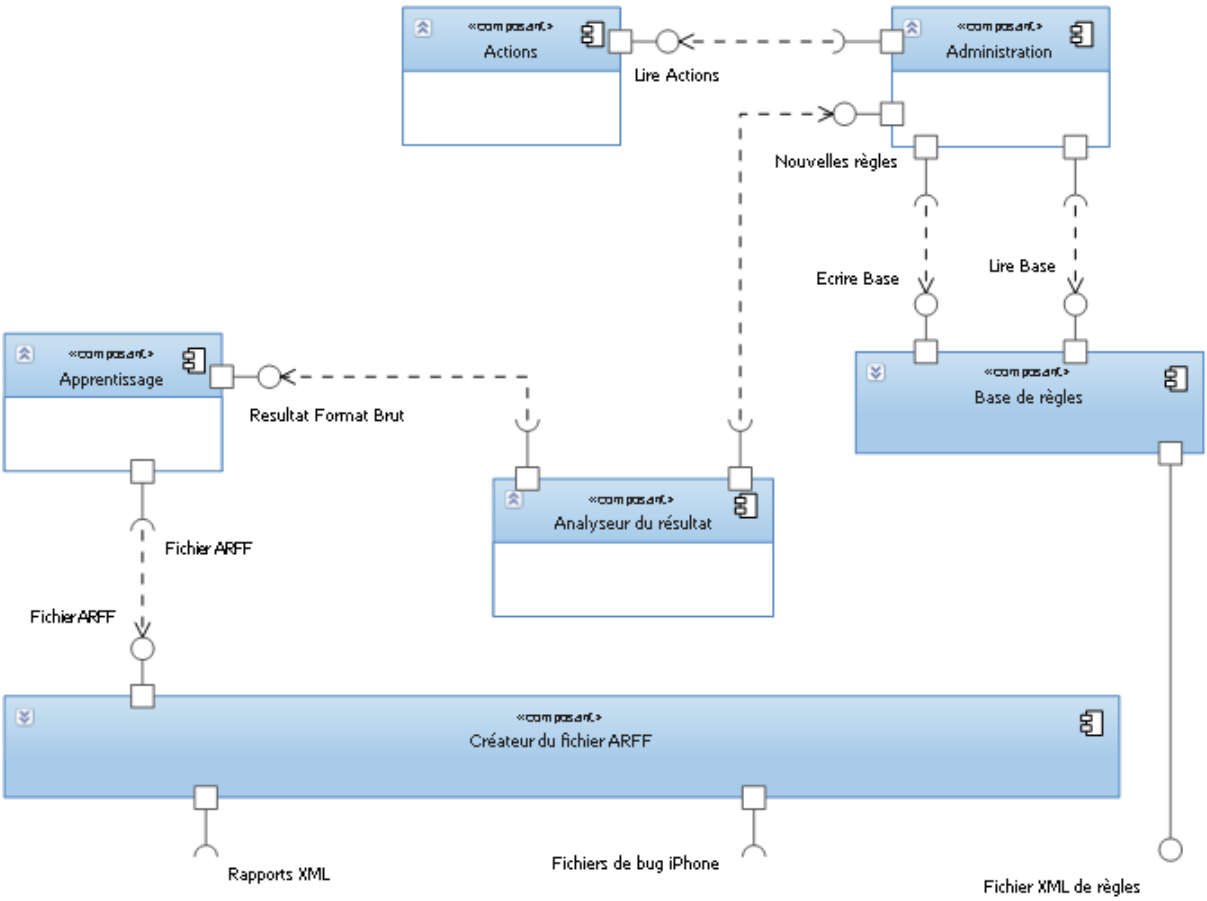

DIAGRAMME 4 - COMPOSANTS SUR LE SERVEUR

La partie création du fichier ARFF que nous avons abstrait est une partie délicate en raison du cas particulier que sont les rapports de mauvais fonctionnement iPhone.

Le diagramme en page suivante développe les trois composants permettant de générer le fichier d'apprentissage. On trouve tout d'abord un lecteur de rapport XML, permettant de récupérer et lire les rapports au format XML. Le générateur de rapports de mauvais fonctionnement iPhone, construit ces rapports sur la base d'un fichier de bug, recoupé avec les rapports de bon fonctionnement produits à un instant proche de la production du rapport de bug. On obtient ainsi les deux types de rapports pour les deux types d'OS mobile supportés.

On rappelle cependant, que si le serveur peut travailler avec les deux OS mobile, du fait de la différence des variables présentes dans les rapports de faits, et la différence naturelle des deux systèmes, le serveur ne fusionne pas des rapports Androïd et des rapports iOS. On a donc deux fichiers ARFF, deux apprentissages, et deux bases de règles au final.

Le compilateur ARFF peut alors générer les jeux d'apprentissage pour chaque type de système mobile sur la base de ces rapports. Ce compilateur doit gérer les attributs manquants (engendrés essentiellement par l'évolution des applications lancées) pour fournir un jeu de test.

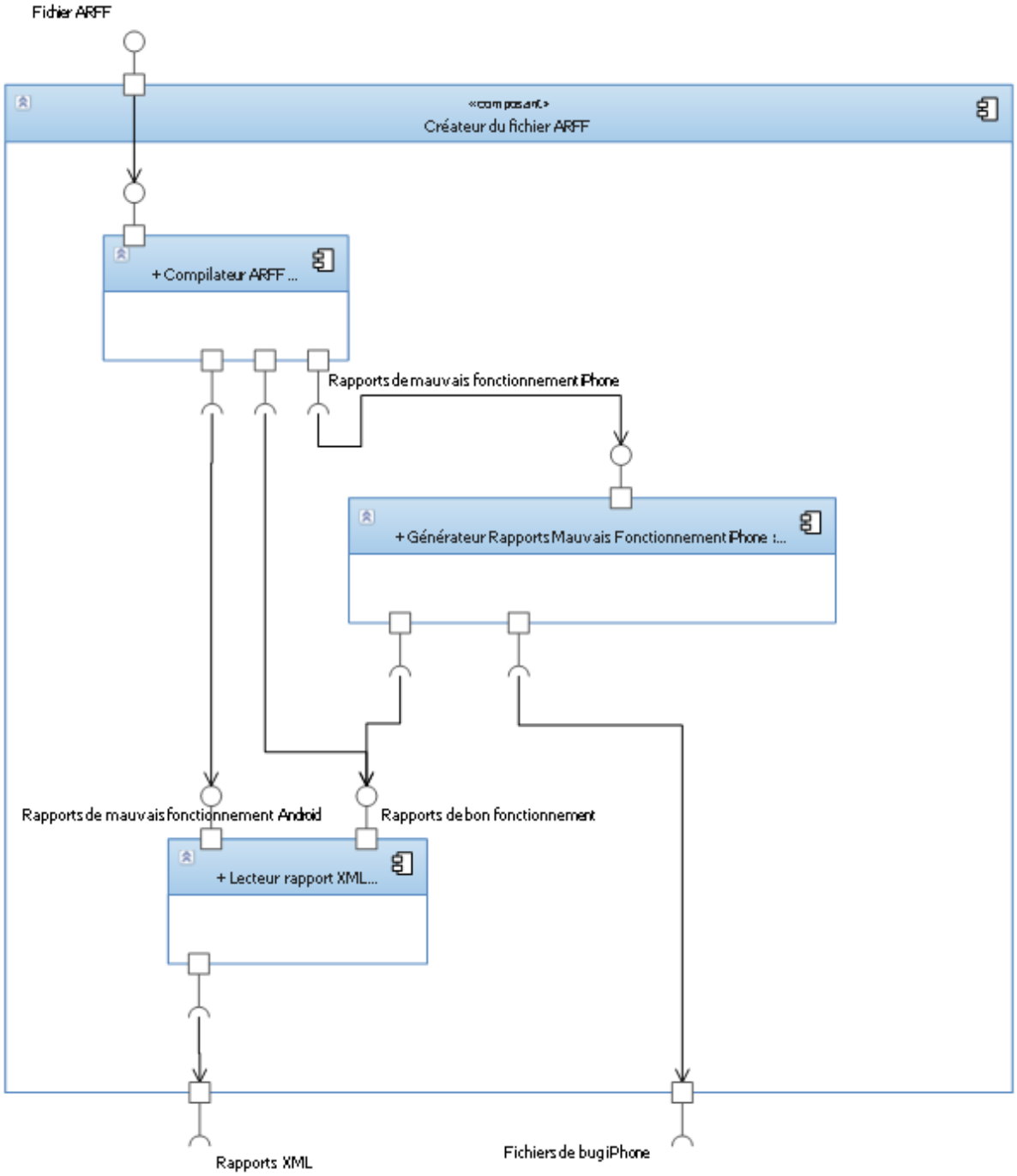

DIAGRAMME 5 - COMPOSANTS DU GENERATEUR DE FICHIERS D'APPRENTISSAGE

2.3. Bases

Comme nous l'avons vu dans les diagrammes précédents, deux bases sont présentes et ont un rôle capital. Nous présenterons ici la structure générale de ces bases. La dynamique

d'exécution (remplissage, génération des rapports, construction…) sera présentée dans les parties suivantes.

2.3.1. Base de faits

La base de faits représente l'état du mobile. C'est un simple ensemble de faits. Un fait peut être de différents types. On retrouve un type par caractéristique/attribut du mobile. Par exemple, la mémoire libre, l'état de l'appareil photo, du vibreur, la version de l'OS, la quantité de mémoire disponible…

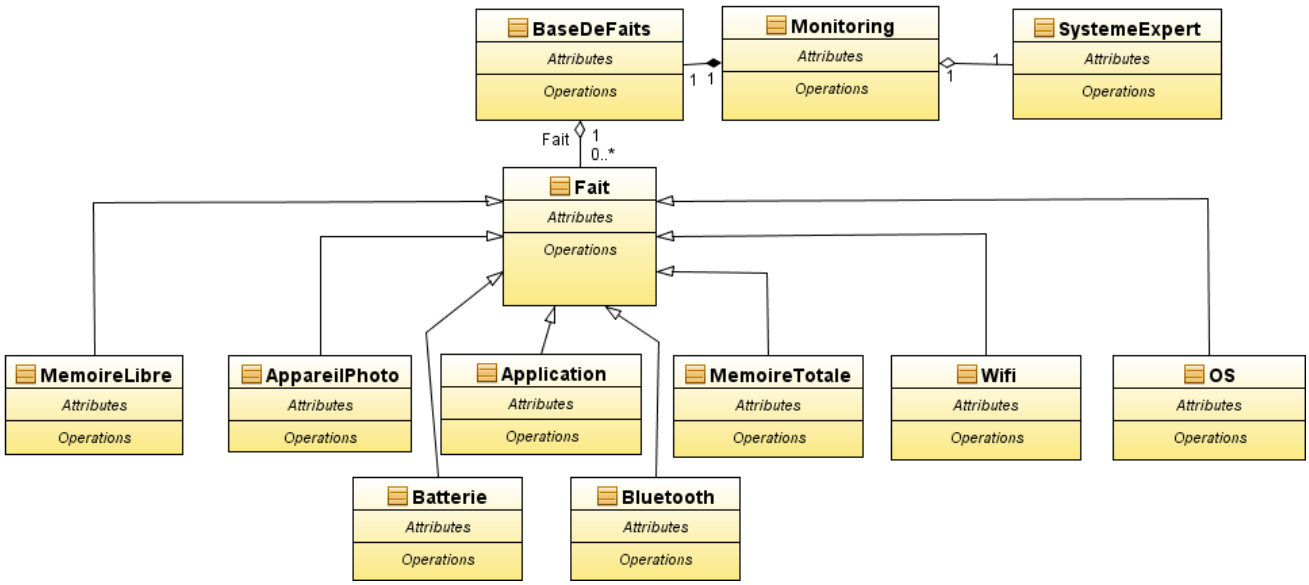

DIAGRAMME 6 - DIAGRAMME DE CLASSE DE LA BASE DE FAITS

2.3.2. Base de règles

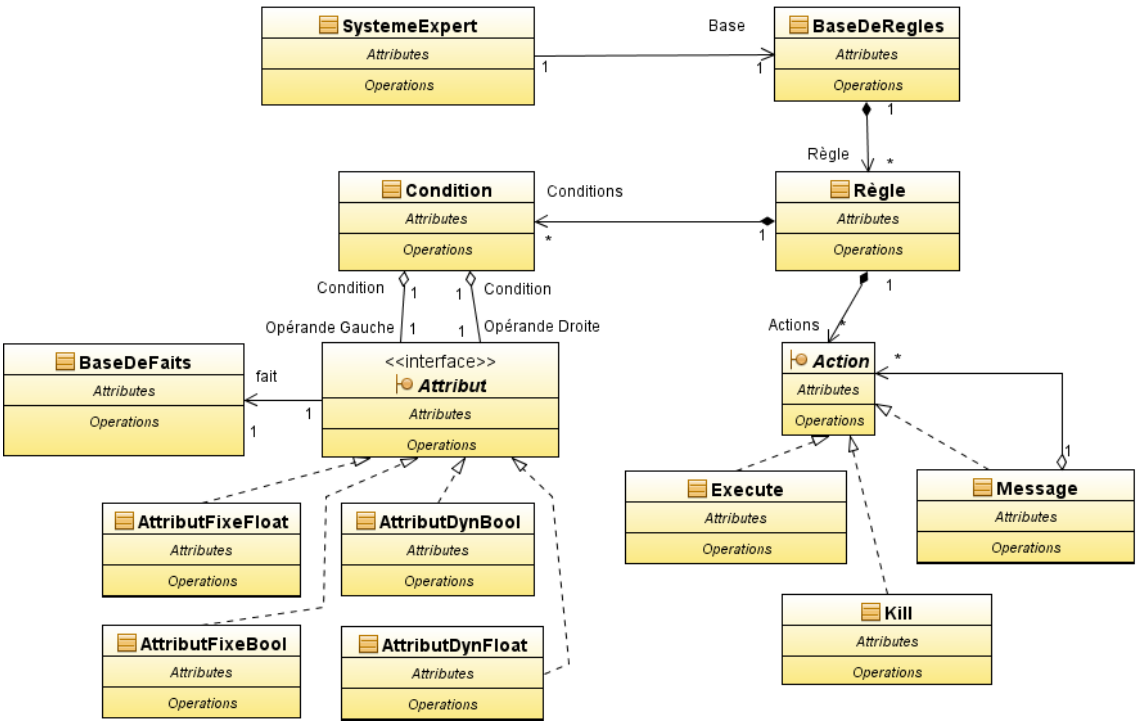

DIAGRAMME 7 - DIAGRAMME DE CLASSE DE LA BASE DE REGLES

La base de règles, présentée sur le diagramme ci-dessus, consiste en un ensemble de règles.

Une règle est elle-même constituée de deux partie :

- Les conditions
- Les actions

Lorsque toutes les conditions sont vérifiées, alors l'action est déclenchée. Par ailleurs, deux règles peuvent avoir une même action.

Les conditions

Une condition est sous la forme :

```
OpérandeGauche [Opérateur] OpérandeDroit
```

Par exemple, on pourrait avoir :

```
MémoireLibre <= 40 // La mémoire libre est inférieure à 40 Mo
Photo = Vibreur    // L'appareil photo et le vibreur sont dans le même état
```

Comme on peut le voir sur les exemples précédents, on retrouve plusieurs types d'opérande droit et gauche :

- **Un type fixe** (true/false pour un booleen et une valeur numérique fixe pour un float)
- **Un type dynamique** (qui fait référence à un fait de la base des faits).

On a donc des attributs soit fixes soit dynamiques et soit float soit booléens. On retrouve donc 4 types d'attributs différents, ce qui permet de représenter toutes les conditions possibles.

Les actions

On retrouve plusieurs actions possibles, comme exécuter un programme, tuer une application ou afficher un message avant d'appliquer d'autres actions.

L'action message est un cas particulier. Le design pattern composite a été utilisé pour permettre d'afficher un message avant d'exécuter une ou plusieurs actions. Ce message pouvant être une simple alerte ou un choix OUI/NON.

3. Partie Mobile

3.1. Système de reporting / Génération des rapports

3.1.1. Données présentes

Les rapports de bon et mauvais fonctionnement vont contenir les données qui nous semblent pertinentes à observer dans le but d'un plantage et qui sont disponibles grâce à l'api de chaque système d'exploitation. Ainsi les données présentent sur iOS et Android seront différentes.

Données Android

- La mémoire totale
- La mémoire libre
- Le système d'exploitation et sa version
- Le Wi-Fi
- La batterie
- Le Bluetooth
- Les applications lancées
- L'appareil Photo

Données iOS

- La mémoire totale
- La mémoire libre
- Le système d'exploitation et sa version
- Les applications lancées
- Le Wifi
- La Batterie

3.1.2. Diagramme de classes

Le système de monitoring possède 4 grandes classes, la classe monitoring qui est le point d'entrée de notre programme, elle possède une base de fait et une référence sur le système expert afin de l'alerter dès qu'il se met à jour.

Pour la version Android, elle va lire les logs et détecter un crash. Dès lors qu'un crash sera détecté, le système de monitoring génèrera un rapport de mauvais fonctionnement.

La base de fait, qui est unique, est composée d'une table de hachage de faits référencé par son nom. Chaque fait possède une valeur, une méthode refresh() qui va mettre à jour la valeur du fait et une méthode toXML() pour écrire le fait dans le rapport XML. Les faits seront les données présentées précédemment.

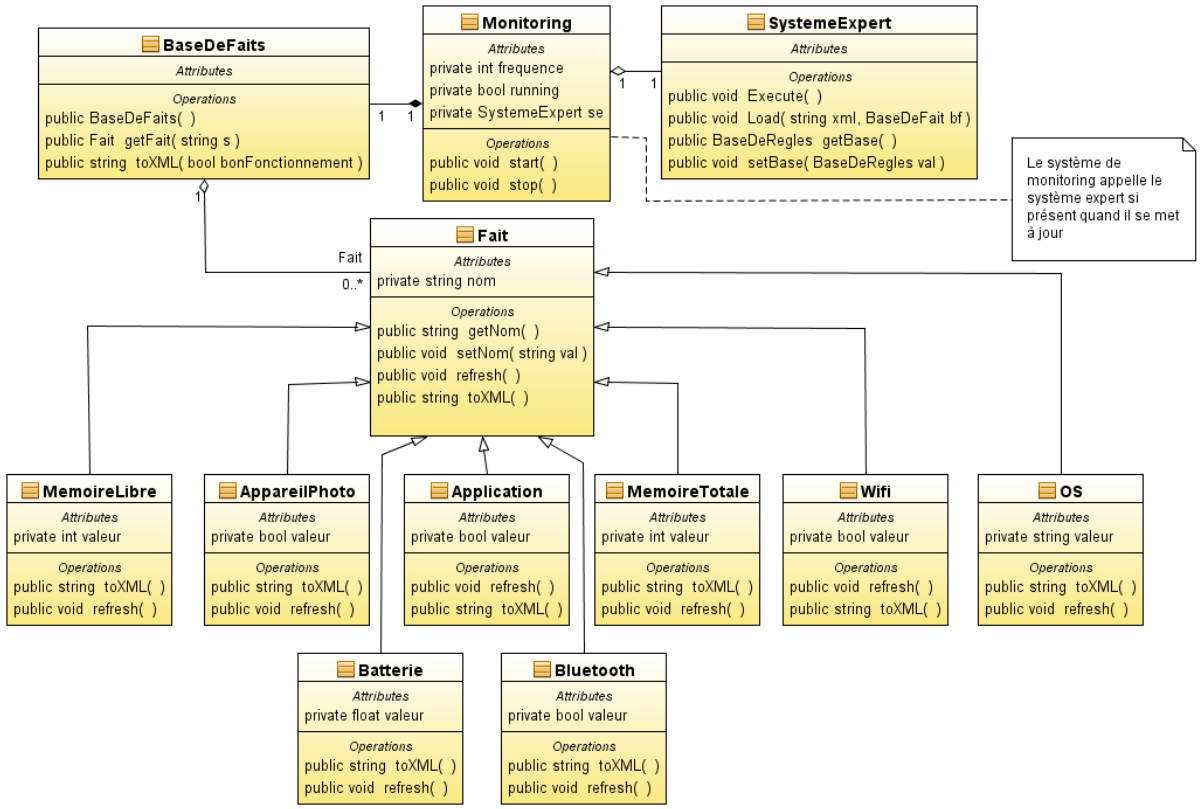

DIAGRAMME 8 DIAGRAMME DE CLASSE DE LA BASE DE FAITS

3.1.3. Diagramme de séquence

Lors de la génération cyclique de rapport de bon fonctionnement la classe monitoring appelle la méthode toXML() de la BaseDeFaits, qui va mettre à jour la valeur de chaque fait et va appeler leur méthode toXML().

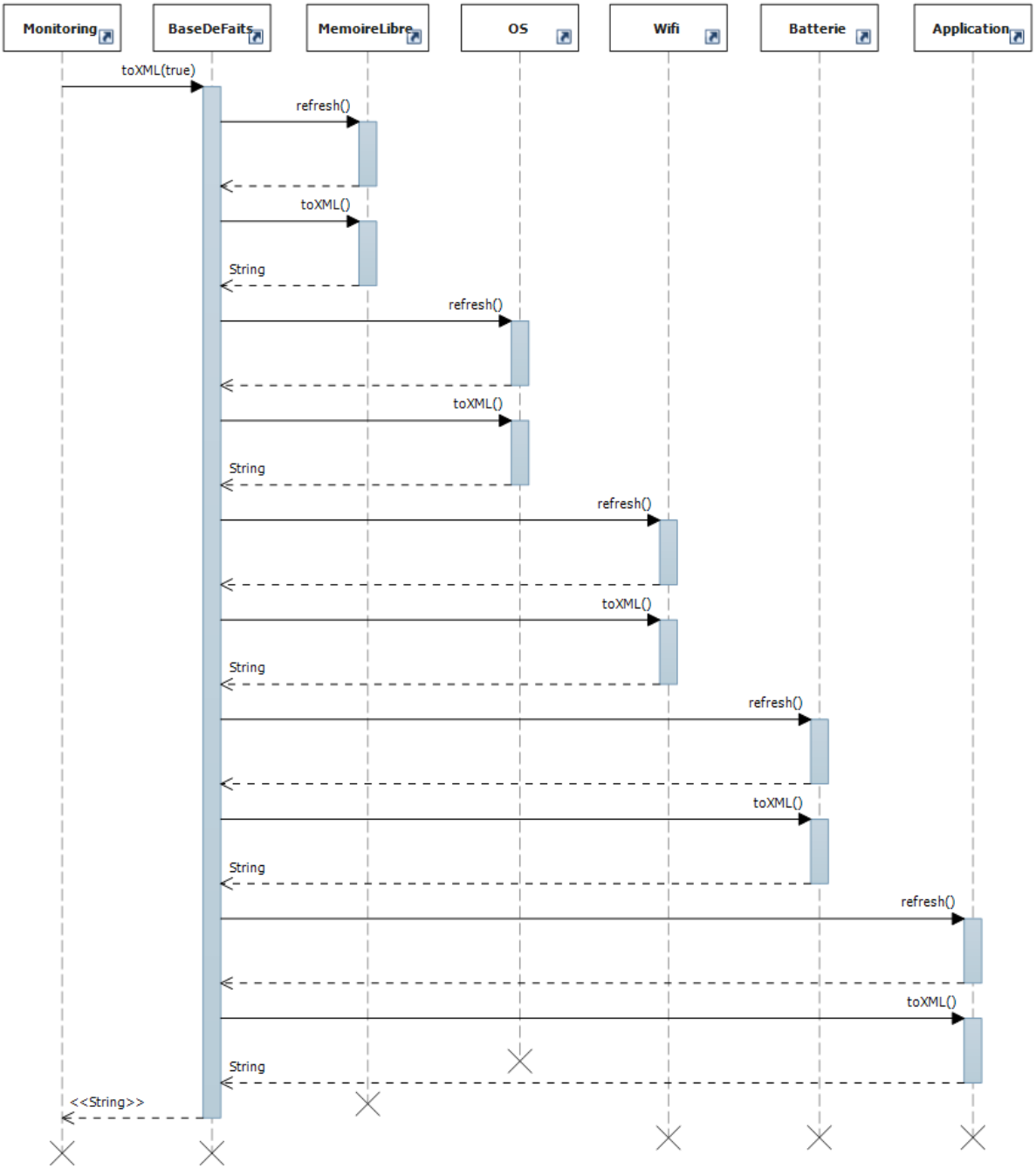

DIAGRAMME 9 GENERATION DE RAPPORT DE BON FONCTIONNEMENT

Dans le cas de la génération de rapport de mauvais fonctionnement du système Android, lorsque la classe monitoring détecte un crash, elle appelle la méthode toXML() qui va appeler la méthode toXML() de chaque fait puis va mettre à jour la valeur du fait.

La mise à jour des valeurs se fait après la génération du rapport pour ne pas avoir des valeurs modifiées par l'action du système expert.

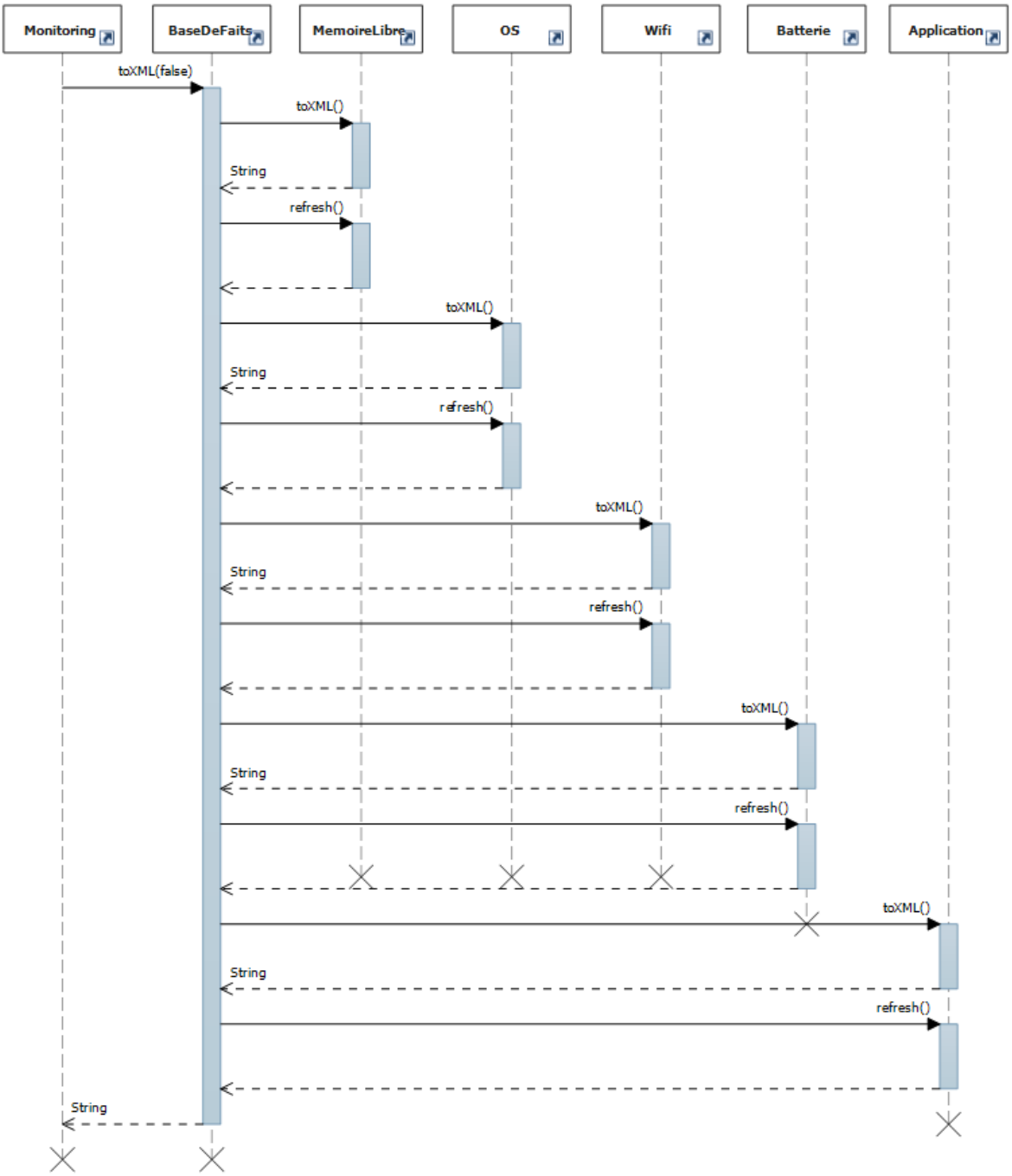

DIAGRAMME 10 GENERATION DE RAPPORT DE MAUVAIS FONCTIONNEMENT

3.2. Système expert

3.2.1. Lecture des règles / Chargement de la base de règles

Le système expert va à partir de la lecture du fichier de règles en XML, charger sa base de règles. Lorsqu'il lira une balise <rule>, la base de connaissance créera une instance de règle, qui lorsque qu'une balise <condition> sera lu, créera une instance de condition, et ainsi de suite.

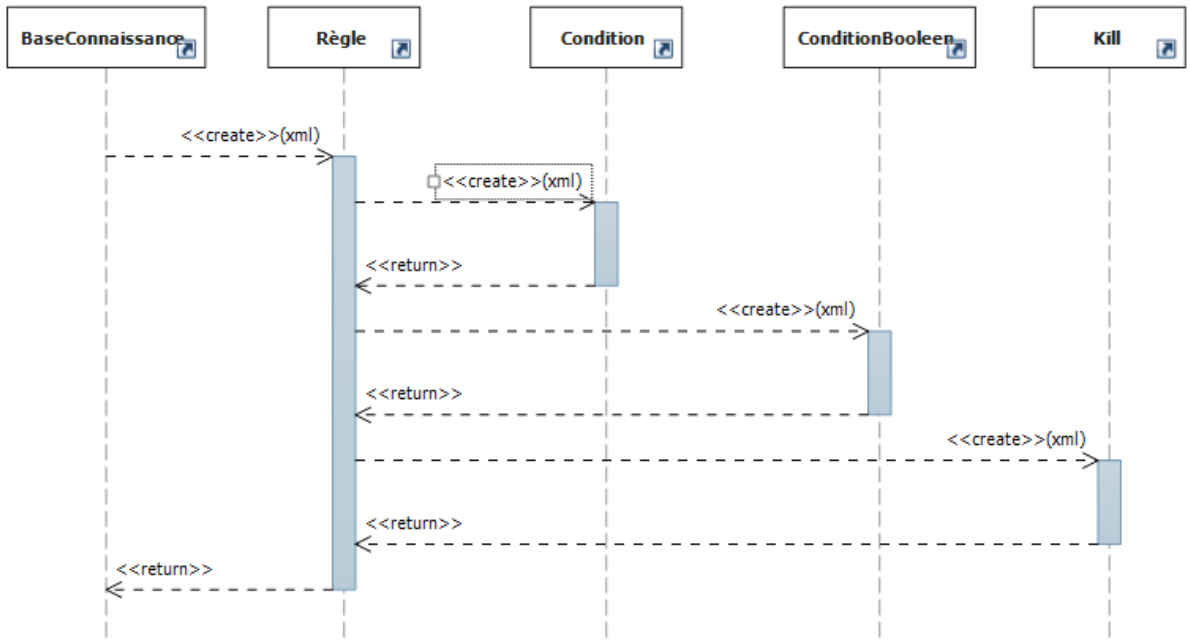

DIAGRAMME 11 CHARGEMENT DE LA BASE DE CONNAISSANCE

3.2.2. Moteur d'inférence

Lorsque la base de fait se met à jour, elle alerte le système expert qui va tenter d'exécuter les règles de la base de règles en regardant si les conditions de chaque règle sont réalisées ou non, et si c'est le cas exécuter l'action de la règle. L'ordre des règles dans la base est important, puisque seule la première règle dans la base qui sera applicable sera appliquée.

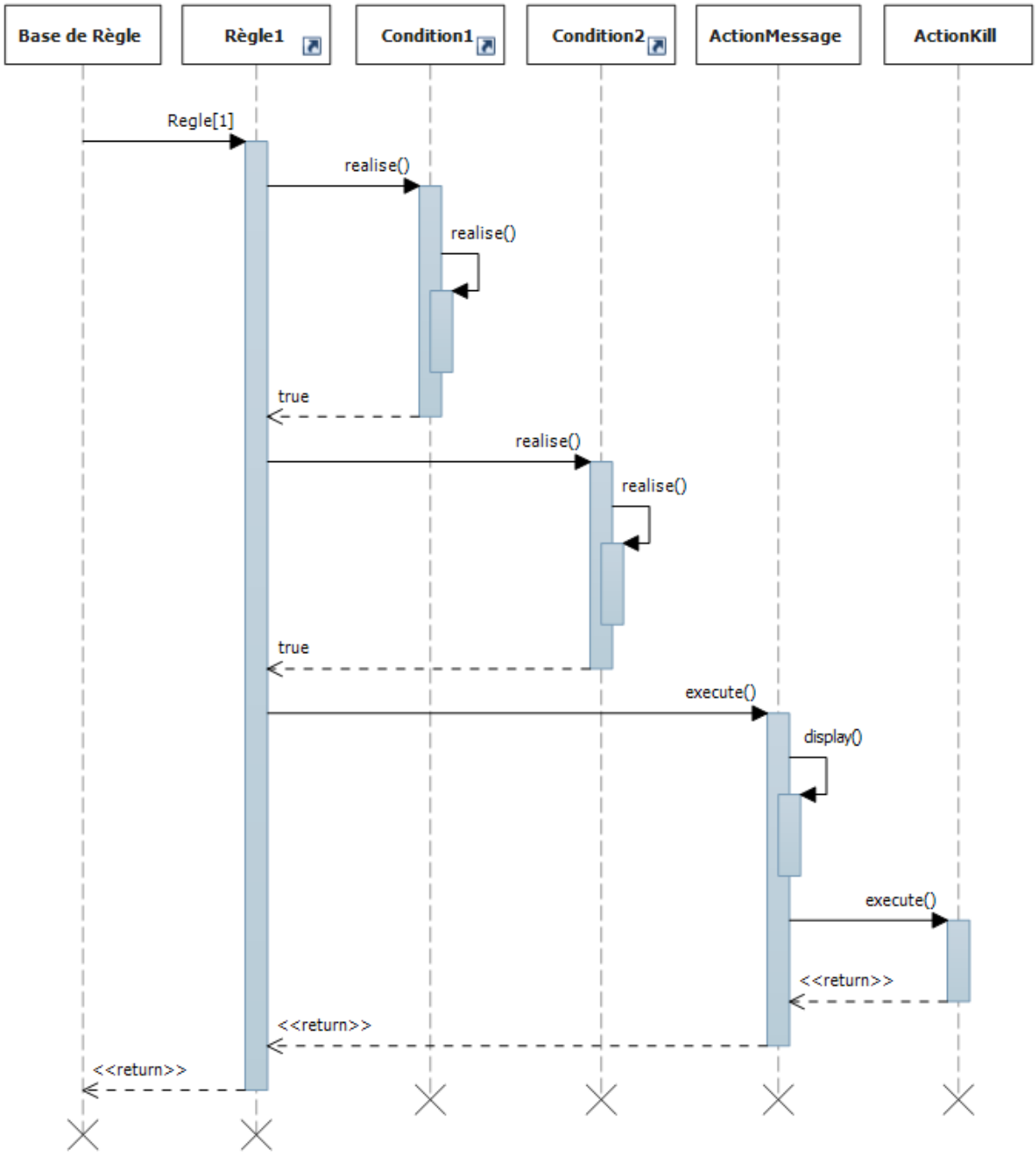

DIAGRAMME 12 INFERENCE DE REGLE

3.2.3. Génération de bilans sur les utilisations des règles

Lors de sa connexion au serveur, le terminal mobile peut envoyer des bilans sur les utilisations des règles. Ces bilans seront générés au format XML. Lors de la génération le Système Expert va appeler la méthode Rapport() de la base de règles, qui va elle-même appeler les méthodes Rapport() de toutes les règles qu'elle possède. Ces méthodes renvoient des chaines de caractères contenant les bilans au format XML.

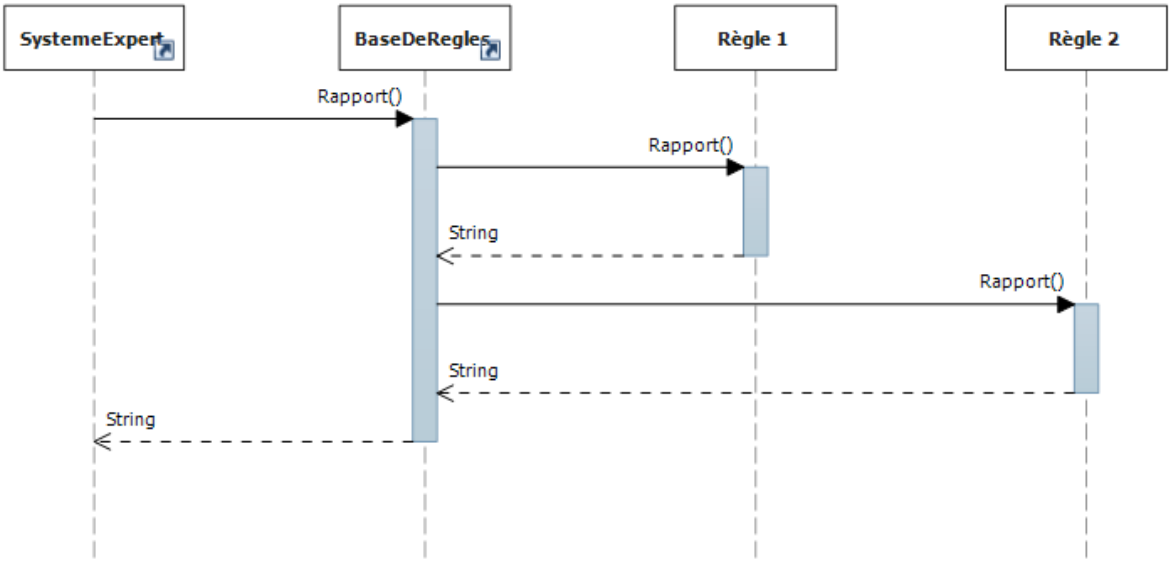

DIAGRAMME 13 GENERATION DES BILANS DE REGLES

3.2.4. Diagramme de classe

Le système expert possède une base de règles qui sera composée d'une table de hachage de règles, référencé par son l'identifiant de la règle. Il sera possible de construire cette base de règles vierge ainsi qu'à partir d'un fichier XML contenant déjà des règles ainsi qu'une base de faits.

Chaque règle sera composée d'une liste de conditions ainsi qu'une liste d'actions. On pourra vérifier si la règle est applicable en appelant la méthode realise(), celle-ci renverra vrai si toute les conditions de la règle sont vérifiées. On pourra alors appeler la méthode execute() qui va exécuter toutes les actions de la règle. De plus chaque règle contiendra un entier représentant le nombre de fois qu'elle a été appliquée.

Une condition est composée d'un opérande gauche, d'un opérateur et d'un opérande droit. Les opérandes sont de 2 types : fixe, c'est-à-dire une constante, ou dynamique, autrement dit une référence sur un fait de la base de faits. La méthode realise() rend vrai si la condition est vérifiée.

Une action peut être de 3 types :

- Kill : terminer une application dont l'identifiant est en attribut
- Execute : lance l'application dont l'identifiant est en attribut
- Message : Permet de demander à l'utilisateur s'il veut appliquer une ou plusieurs actions (Kill ou Execute) ou non

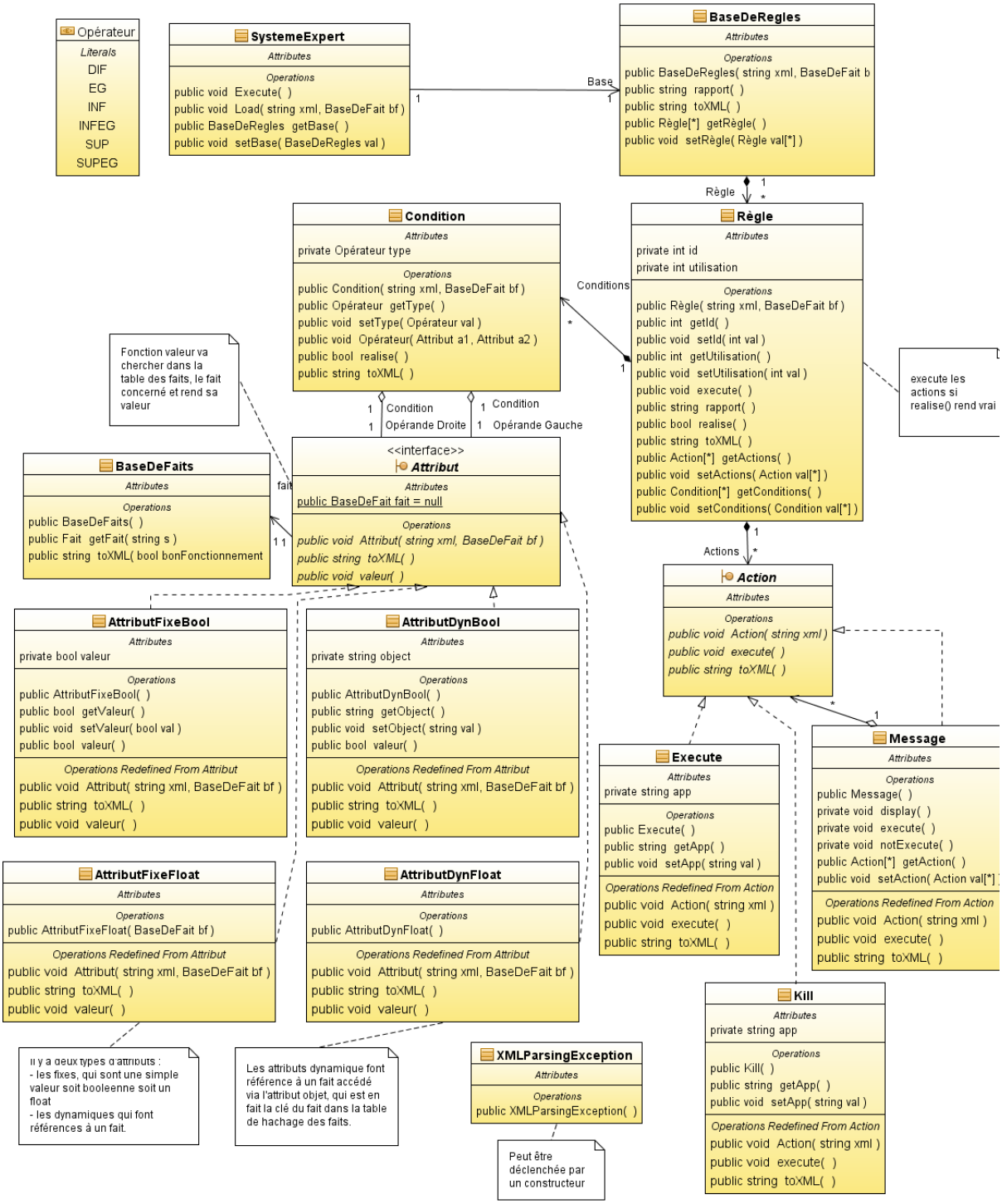

DIAGRAMME 14 DIAGRAMME DE CLASSE DE LA BASE DE REGLES

3.3. Suivi de l'utilisateur

Le suivi utilisateur sera chargé d'enregistrer les actions effectuées par l'utilisateur sur son appareil mobile.

Pour cela, on utilisera une classe héritant de Service (représentant un service Android), qui, lors de sa création, lancera une tâche asynchrone chargée de récupérer les évènements liés au smartphone.

Ces derniers seront obtenus à l'aide de l'utilitaire « logcat » intégré au système Android.

Pour des raisons d'espace disque, on ne pourra pas stocker indéfiniment toutes les évènements détectés, il faudra donc penser à un système de file où l'on retire le premier élément quand la taille de cette dernière devient trop importante.

L'enregistrement des logs se fera au format XML afin de permettre une réutilisation plus aisée par la suite. Ce fichier sera sous la forme suivante :

```
<?xml version="1.0" encoding="UTF-8" ?>
<log>
    <entry time=”TIME” type=”TYPE” desc=”DESCRIPTION”>
</log>
```

TIME : Entier représentant la date

TYPE : Type de l'évènement, par exemple :

- INCOMINGCALL : Appel reçu
- CALL : Appel émis
- RUN : Application lancée
- CLOSE : Application fermée
- CRASH : Application ayant planté

DESCRIPTION : Description de l'évènement. Pour un appel ce sera par exemple le numéro appelé ou le numéro appelant, pour une application lancée, on indiquera le nom de cette dernière.

Voici un exemple de fichier XML à obtenir en sortie :

```
<?xml version="1.0" encoding="UTF-8" ?>
<log>
    <entry time=”1292358480” type=”CALL” desc=”+33583953502”>
    <entry time=”1292358590” type=”RUN” desc=”com.insa.applicationtest”>
    <entry time=”1292358480” type=”CLOSE” desc=”com.google.gmail”>
    <entry time=”1292358480” type=”CALL” desc=”+33693155522”>
</log>
```

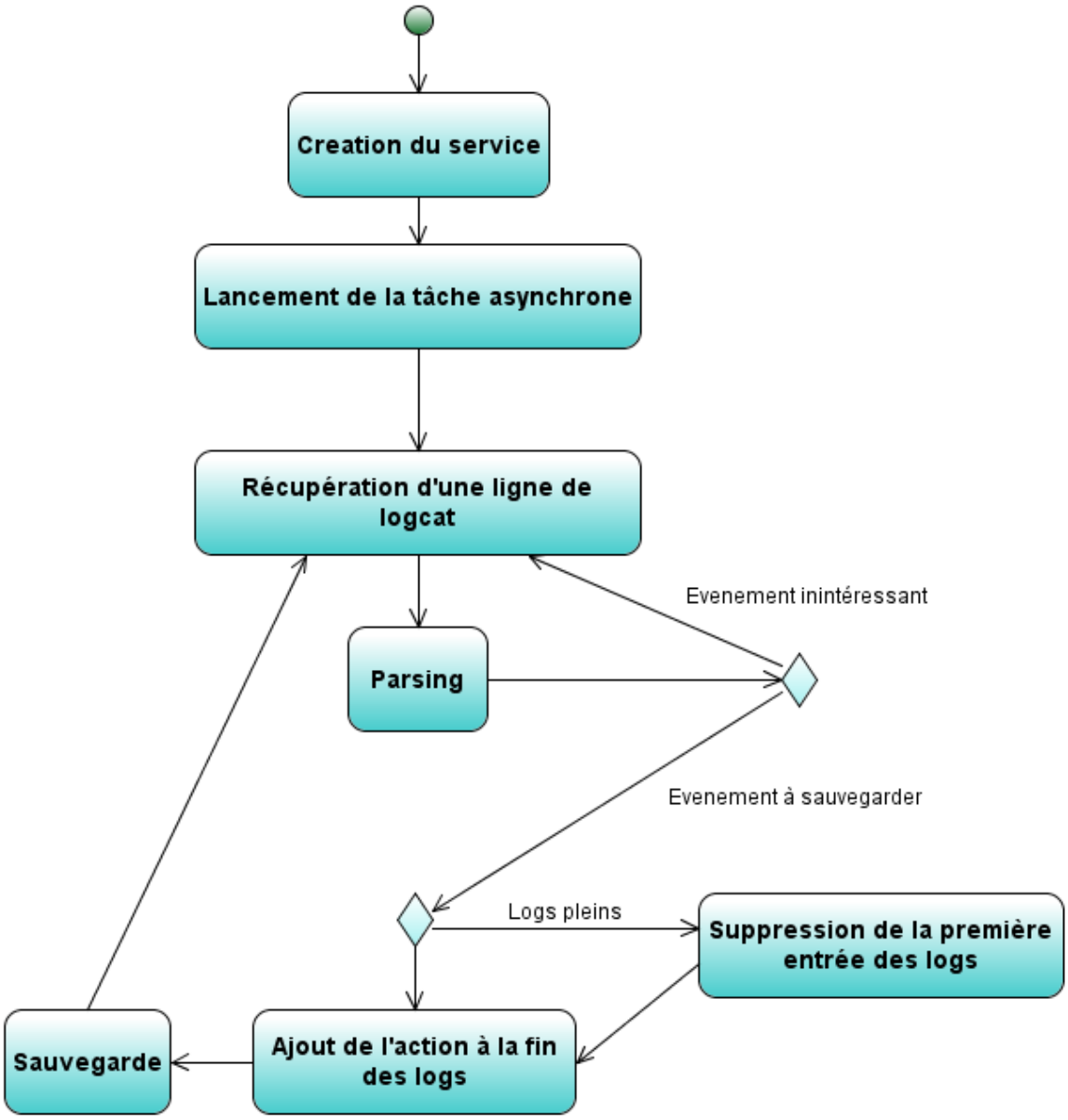

DIAGRAMME 15 DIAGRAMME D'ACTIVITE CORRESPONDANT A L'EXECUTION DU SUIVI UTILISATEUR

Dans le diagramme d'activité ci-dessus, il est possible de voir que l'on crée tout d'abord le service chargé de récupérer les actions de l'utilisateur. Ce dernier va lancer une tâche asynchrone, permettant son exécution en arrière plan, qui utilisera l'utilitaire Android « logcat » retournant des lignes de log contenant diverses informations à propos de l'état du téléphone. Pour chaque ligne récupérée, on vérifiera si l'évènement est ou non intéressant, et si l'on doit le conserver. Si c'est le cas, on l'ajoute à un ensemble d'évènements enregistrés, en prenant le soin de supprimer l'évènement le plus ancien si l'on en a déjà sauvegardé plus d'un certain nombre. Finalement, on sauvegarde les logs dans un fichier, puis l'on continue l'exécution de la tâche asynchrone.

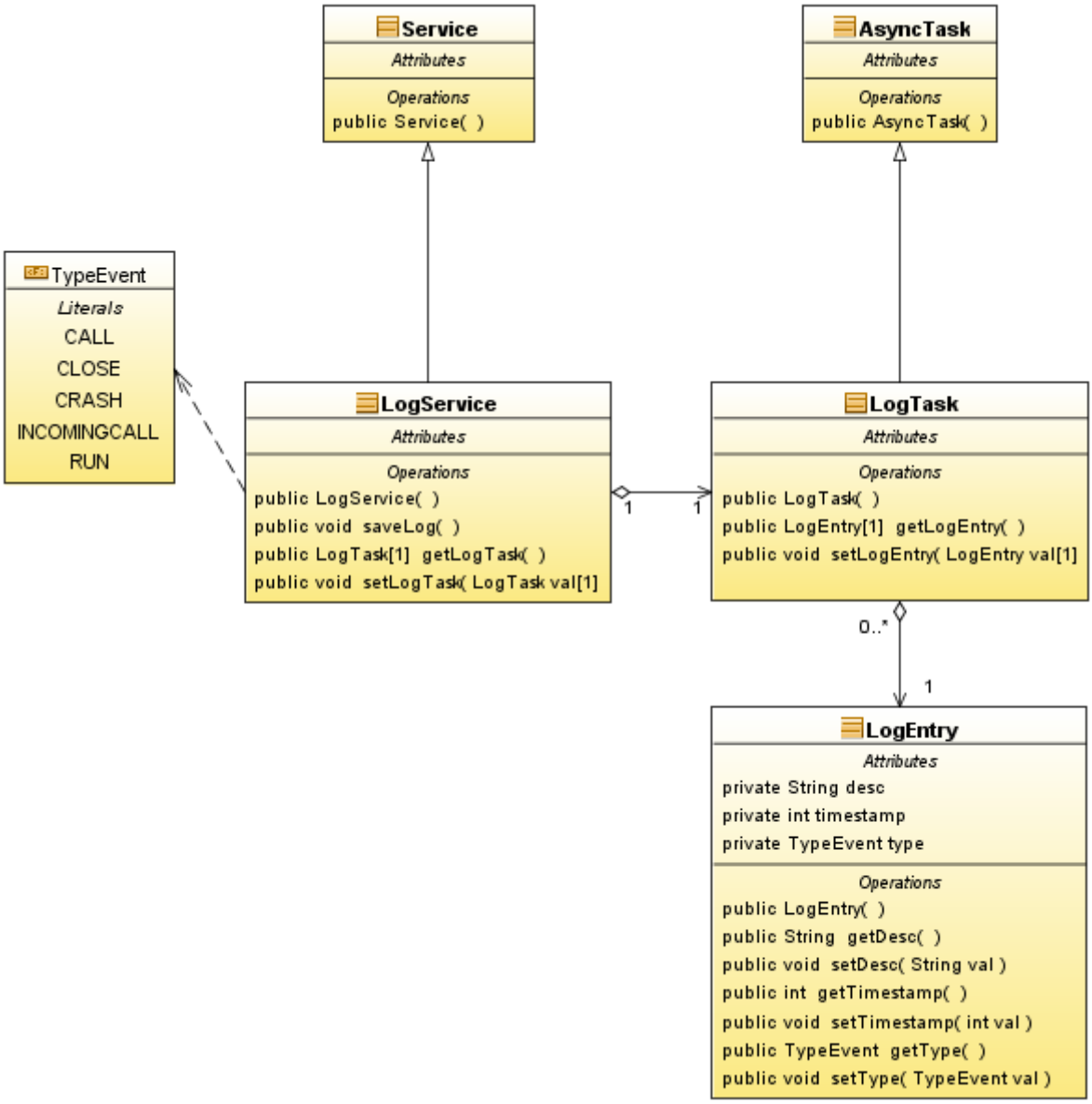

DIAGRAMME 16 DIAGRAMME DE CLASSES DU SUIVI UTILISATEUR

Ici, Service et AsyncTask sont des classes définies dans le SDK Android permettant de créer respectivement un service, et une tâche asynchrone. On crée donc deux classes héritant de ces dernières afin d'obtenir un nouveau type de service et un nouveau type de tâche asynchrone.

LogService sera donc chargée de créer et lancer une instance de LogTask, qui elle, à son tour, exécutera logcat, et créera des instances de la classe LogEntry (correspondant à des évènements sauvegardés).

Les types d'évènements que l'on conserve sont listés dans l'énumération TypeEvent.

4. Partie Serveur

4.1. Compilation

4.1.1. Création des rapports XML

Sous Android, la création des rapports de bon et mauvais fonctionnement au format XML est assurée directement par le smartphone (en détectant nous même les plantages), en revanche pour l'iPhone, il n'est pas possible de récupérer directement les rapports de mauvais fonctionnement dans ce format, les informations concernant les crash étant stockées seulement dans des crash logs. Il faut donc traiter ces derniers, ne contenant pas forcément toutes les informations nécessaires, en les recoupant avec les rapports de bon fonctionnement.

Ce compilateur prend donc en entrée des rapports de bon fonctionnement iPhone, ainsi que des rapports de crash iPhone. Il sera alors chargé de convertir ces derniers sous forme de rapports XML valides.

Le problème ici provient du fait que :

- Les rapports de crash ne sont pas sous le bon format
- Certains attributs sont manquants et doivent être renseignés

Pour pouvoir effectuer cette conversion, on procédera en deux étapes. Tout d'abord le compilateur se chargera de lire les rapports de bon fonctionnement, et les rapports de mauvais fonctionnement afin de les stocker sous un type d'objet similaire (classe Rapport). On en profitera également pour sauvegarder les différents attributs rencontrés durant la lecture (objet mAttributsExistants de la classe Compilateur). Une fois cette phase terminée, on obtient normalement des rapports dont certains attributs sont absents, mais également une liste complètement des attributs que les rapports peuvent contenir.

Cela étant fait, la deuxième étape est chargée de remplir ces attributs manquants. Pour cela, elle va parcourir tous les objets Rapport créés par le compilateur, et traiter les attributs des rapports ayant indiqués comme des rapports de crash. Dans ces derniers, on va ajouter les attributs manquants (c'est-à-dire étant dans l'objet mAttributsExistants du Compilateur, mais pas dans le rapport de crash), la valeur sera mise en fonction de la possibilité que le compilateur

a de la deviner. Si le rapport précédent et suivant possèdent cet attribut et qu'ils ont les mêmes valeurs, on mettra une valeur identique pour le rapport de crash. Dans le cas contraire, on mettre la valeur à « ? ».

Lorsque tous les rapports auront été traités, on pourra sauvegarder chacun d'entre eux sous le format XML spécifique aux rapports (normalement identique au format d'entrée pour les rapports de bon fonctionnement puisqu'on ne les aura pas modifiés), sans oublier d'ajouter la balise <crash /> pour les rapports de bug.

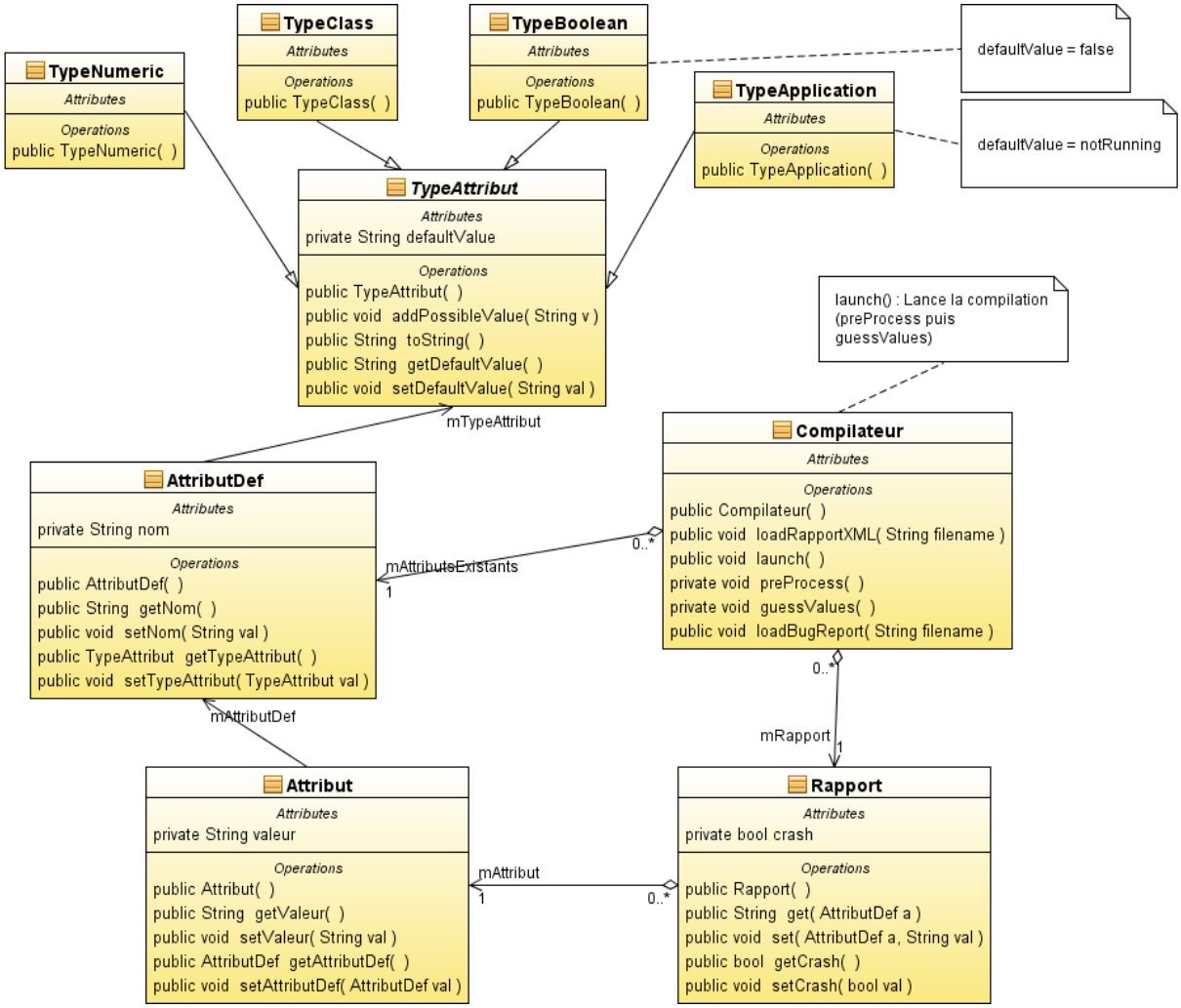

DIAGRAMME 17 DIAGRAMME DE CLASSE DU COMPILATEUR CREANT DES RAPPORTS AU FORMAT XML

Dans le diagramme ci-dessus, la classe principale est celle nommée Compilateur. Elle contient une fonction launch(), lançant la compilation, qui appellera tout d'abord la méthode preProcess() effectuant la phase de prétraitement décrite précédemment, puis la méthode guessValues(), chargée de retrouver les valeurs manquantes des rapports de crash.

Lors de la phase de prétraitement, le compilateur chargera tous les rapports en créant des instances de la classe Rapport et créera en même temps, pour chaque nouveau type d'attribut rencontré, une instance d'AttributDef , permettant de recenser les attributs existants.

Pour chaque objet Rapport, on aura une liste d'objet Attribut correspondant aux attributs qui lui sont liés. Les différents types d'attributs possibles héritent de la classe TypeAttribut et peuvent être de type numérique (TypeNumeric), une énumération de valeurs (TypeClass), de type booléen (TypeBoolean), ou décrire l'état d'une application (TypeApplication).

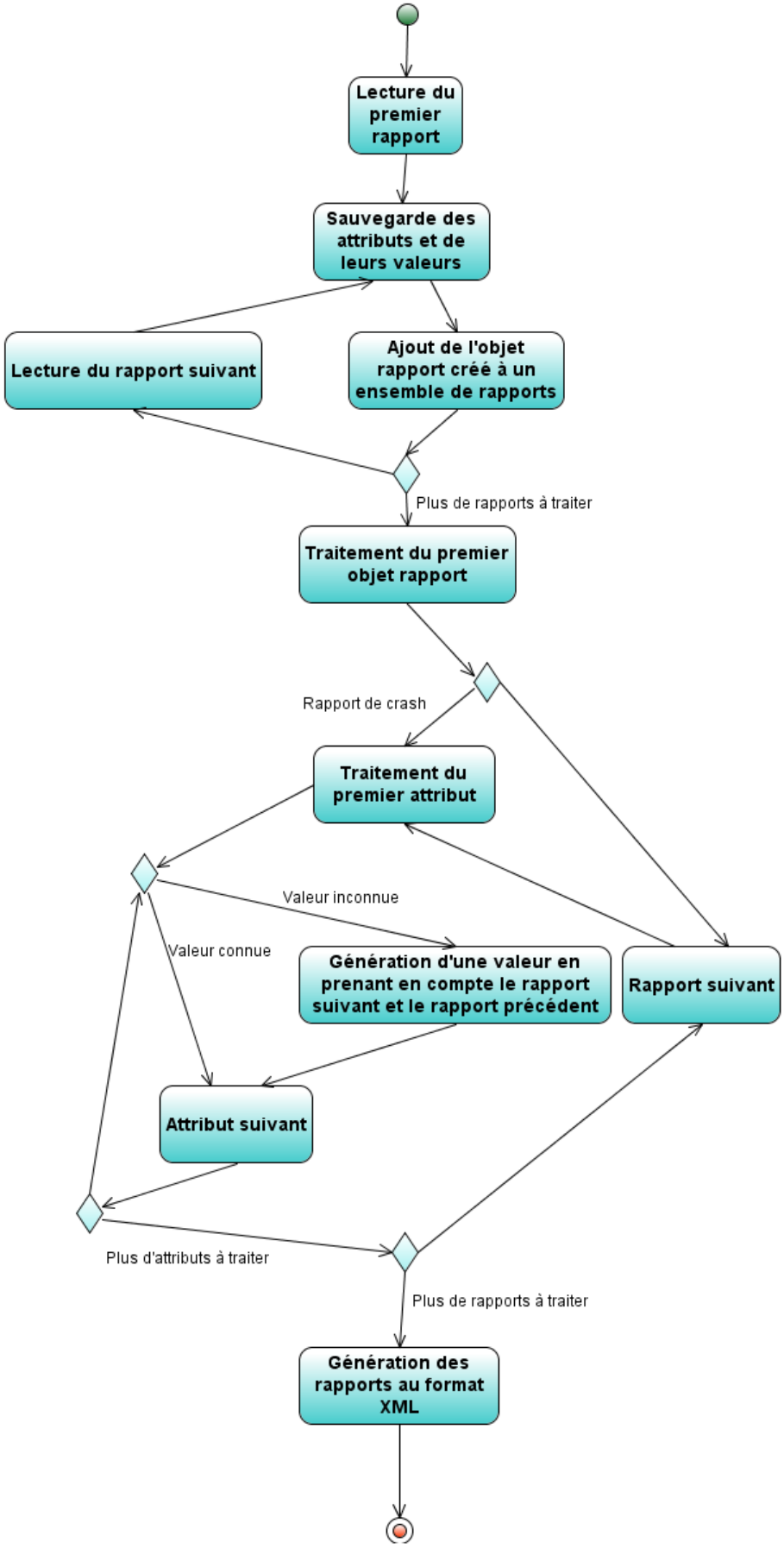

DIAGRAMME 18 DIAGRAMME D'ACTIVITE DE LA CREATION DE RAPPORTS AU FORMAT XML

Dans le diagramme d'activité ci-dessus, on peut voir que le compilateur va tout d'abord dans un premier temps lire tous les rapports et sauvegarder les attributs et leurs valeurs. Puis dans un deuxième temps, traiter tous les rapports de crash afin de remplir les attributs manquants dans chaque rapport. Finalement, il génèrera tous les rapports au format XML.

4.1.2. Compilateur XML vers ARFF

Ce compilateur prend en entrée des rapports de bons et mauvais fonctionnement iPhone et Android et a pour but de les convertir en fichier au format ARFF. Il fonctionne sur le même principe que le compilateur précédent mais ne prend pas en charge les rapports de mauvais fonctionnement iPhone. La différence sera tout d'abord que la deuxième phase (chargée de deviner les attributs manquants) sera effectuée sur tous les rapports (et non plus seulement sur les rapports de crash), et qu'on obtiendra en sortie un seul et unique fichier .ARFF comportant les données de tous les rapports.

On aura donc tout d'abord une première étape de prétraitement où l'on chargera tous les rapports de bons et mauvais fonctionnement au format XML afin de les stocker sous forme d'objets Rapport et également, de recenser tous les attributs existants. A la fin de cette étape, l'en-tête du fichier .ARFF (contenant les différents types d'attributs) pourra être généré à l'aide de la liste des attributs rencontrés.

La deuxième étape aura pour but d'initialiser les attributs manquants des différents rapports, soit si c'est possible, en fonction des rapports précédents et suivants, soit en mettant leur valeur à une valeur par défaut (par exemple « notrunning » pour les attributs décrivant l'état des applications).

Une fois l'opération guessValues() (chargée de deviner les valeurs des attributs manquants d'un rapport) terminée, le compilateur créera un unique fichier au format ARFF avec tous les attributs renseignés pour chaque rapport.

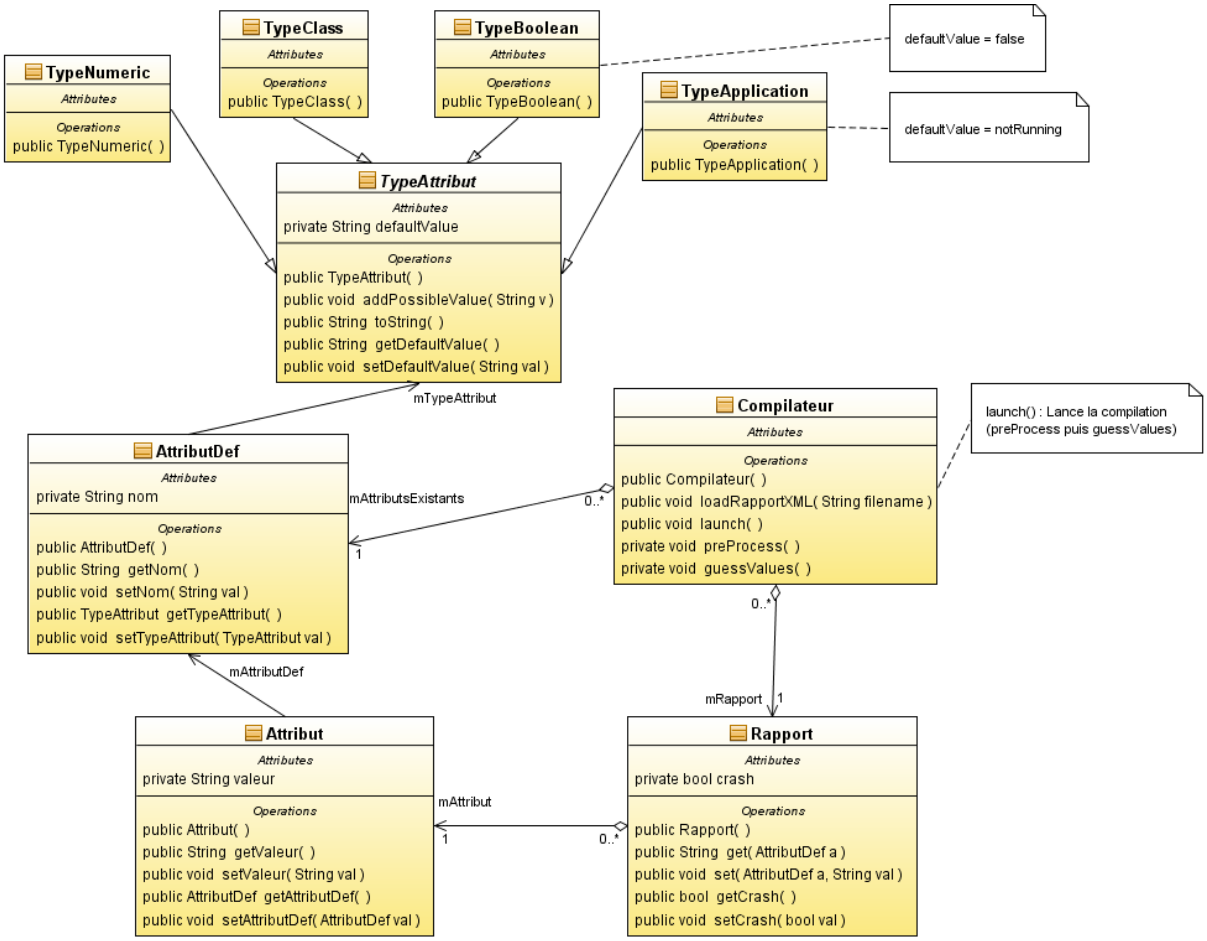

DIAGRAMME 19 DIAGRAMME DE CLASSE DU COMPILATEUR XML → ARFF

Dans le diagramme de classes ci-dessus, les classes présentes sont identiques à celles du compilateur transformant les rapports de crash de l'iPhone au format XML. On peut noter comme différence l'absence de la fonction dans la classe Compilateur permettant le chargement des rapports de crash, mais la majorité des changements se retrouvera dans le code de compilateur.

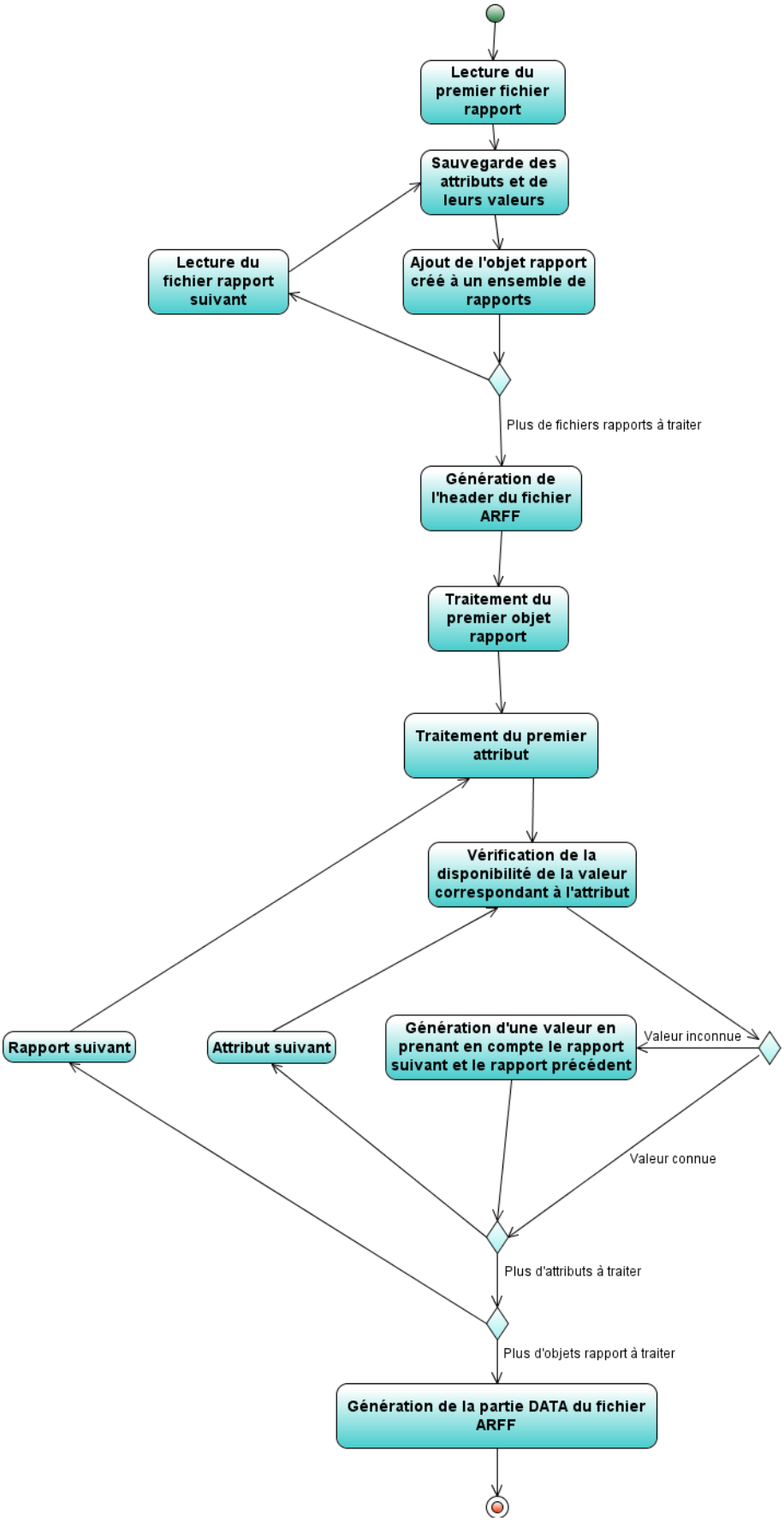

DIAGRAMME 20 DIAGRAMME D'ACTIVITE DU COMPILATEUR XML → ARFF

Dans ce diagramme d'activité, on retrouve globalement les mêmes étapes que dans le diagramme d'activité du compilateur créant les rapports au format XML. Toutefois on peut remarquer l'insertion d'une étape permettant de créer les en-têtes du fichier ARFF, ainsi que le fait que dans la deuxième phase, tous les rapports sont pris en compte, contrairement à l'autre compilateur où l'on traitait seulement les rapports de crash. Finalement la dernière étape ne créera plus tous les rapports au format XML, mais sera chargée de générer la partie DATA du fichier ARFF.

4.2. Apprentissage

Pour l'apprentissage, il est nécessaire de convertir le format de sortie du logiciel Weka en un format plus aisé à manipuler, ici au format XML. La sortie étant différente en fonction de l'algorithme d'apprentissage utilisé, les explications suivantes sont valables seulement pour l'algorithme J48 (arbres de décisions).

Voici un exemple de conversion :

Entrée	Sortie
attribut1 = val1 | attribut2 = val2 | | attribut3 = val3 : res1 (1.0) | | attribut4 = val4 : res2 (1.0) | attribut5 = val5 | | attribut6 = val6 | | | attribut7 = val7 : res1 (1.0) | | | attribut8 = val8 : res2 (1.0) | | attribut9 = val9 : res3 (1.0) attribut10 = val10 : res1 (1.0)	<?xml version="1.0" encoding="UTF-8" ?> <rules> <attribut name="attribut1" Valeur="val1"> <attribut name="attribut2" Valeur="val2"> <attribut name="attribut3" Valeur="val3"> <then>res1</then> </attribut> <attribut name="attribut4" Valeur="val4"> <then>res2</then> </attribut> </attribut> <attribut name="attribut5" Valeur="val5"> <attribut name="attribut6" Valeur="val6"> <attribut name="attribut7" Valeur="val7"> <then>res1</then> </attribut> <attribut name="attribut8" Valeur="val8"> <then>res2</then> </attribut> </attribut> <attribut name="attribut9" Valeur="val9"> <then>res3</then> </attribut> </attribut> </attribut> <attribut name="attribut10" Valeur="val10"> <then>res1</then> </attribut> </rules>

Les règles en entrée du compilateur (générées par WEKA) sont au format suivant :

```
<Attribut> = <Valeur>
(|* <Attribut> = <Valeur>)*: <Valeur>
```

Pour générer le fichier XML, on procédera donc de la façon suivante :

- Inclusion des lignes d'en-têtes (<xml …> <rules>)
- Lecture ligne par ligne de la sortie et remplacement par l'équivalent en XML en utilisant des expressions régulières

Entrée	Sortie
(|[\s]*)*<Attribut> = <Valeur>\n	`<attribut name="Attribut2" Valeur="Valeur2">`

Entrée	Sortie

(|[\s]*)^{n}<Attribut> = <Valeur> : <Valeur2> ([0-9/.]*)	`<attribut name="Attribut" Valeur="Valeur">`
	`<then>Valeur2</then>`
	`</attribut>`

Dans ce dernier cas, il faut compter le nombre d'occurrences de (|[\s]*) afin de savoir le niveau dans lequel on se situe dans l'arbre. A chaque lecture de ligne on stocke donc dans une variable ce nombre, et à la ligne suivante on calcule la différence N entre l'ancienne valeur et la nouvelle valeur.

Si cette dernière est supérieure à 0, on ajoute N fois la balise </attribut> avant d'insérer les nouvelles données.

Inclusion des lignes de fin de fichier, sans oublier de remettre le compteur à 0 et d'ajouter N balises </attribut>.

Exemple :

Dans ce tableau « Anc. Val » et « Nouv. val » correspondent respectivement à l'ancienne et à la nouvelle valeur de N à chaque étape. E correspond au numéro d'étape.

E	Entrée	Anc. val.	Nouv. val.	Diff	Sortie

1	attribut1 = val1	0	0	0	`<?xml version="1.0" encoding="UTF-8" ?>`
					`<rules>`
					`<attribut name="attribut1" Valeur="val1">`

2	| attribut2 = val2	0	1	-1	`<attribut name="attribut2" Valeur="val2">`

3	| | attribut3 = val3 : res1 (1.0)	1	2	-1	`<attribut name="attribut3" Valeur="val3">`
					`<then>res1</then>`
					`</attribut>`

4	| | attribut4 = val4 : res2 (1.0)	2	2	0	`<attribut name="attribut4" Valeur="val4">`
					`<then>res2</then>`
					`</attribut>`

5	| attribut5 = val5	2	1	1	`</attribut>` `<attribut name="attribut5" Valeur="val5">`

6	| | attribut6 = val6	1	2	-1	<attribut name="attribut6" Valeur="val6">
7	| | | attribut7 = val7 : res1 (1.0)	2	3	-1	<attribut name="attribut7" Valeur="val7"> <then>res1</then> </attribut>
8	| | | attribut8 = val8 : res2 (1.0)	3	3	0	<attribut name="attribut8" Valeur="val8"> <then>res2</then> </attribut>
9	| | attribut9 = val9 : res3 (1.0)	3	2	1	</attribut> <attribut name="attribut9" Valeur="val9"> <then>res3</then> </attribut>
10	attribut10 = val10 : res1 (1.0)	2	0	2	</attribut> </attribut> <attribut name="attribut10" Valeur="val10"> <then>res1</then> </attribut>
11	<EOF>	0	0	0	</rules>

Lors de l'étape 1, on génère l'en-tête du fichier XML ainsi que la ligne correspondant à la première valeur en entrée. Lorsque l'on passe à la deuxième, on rencontre une fois le symbole « | », ce qui signifie que l'on est descendu d'un niveau dans l'arbre, on change donc la valeur de N en conséquence. La différence entre l'ancienne valeur et la nouvelle étant négative, on insère seulement la ligne correspondant à la valeur en entrée.

A l'étape 5, la nouvelle valeur que l'on attribue à N est égale à 1, l'ancienne étant égale à 2, on a une différence de 1. Cela signifie qu'il faut insérer une fois une balise fermante </attribut> avant d'insérer les lignes correspondant à notre ligne en entrée. On retrouve ce même raisonnement a l'étape 9 et à l'étape 10 (où l'on insère deux balises </attribut>).

Finalement, à l'étape 11, la fin de fichier étant atteinte, on insère la balise fermante </rules> afin d'indiquer la fin du fichier XML.

4.3. Administration

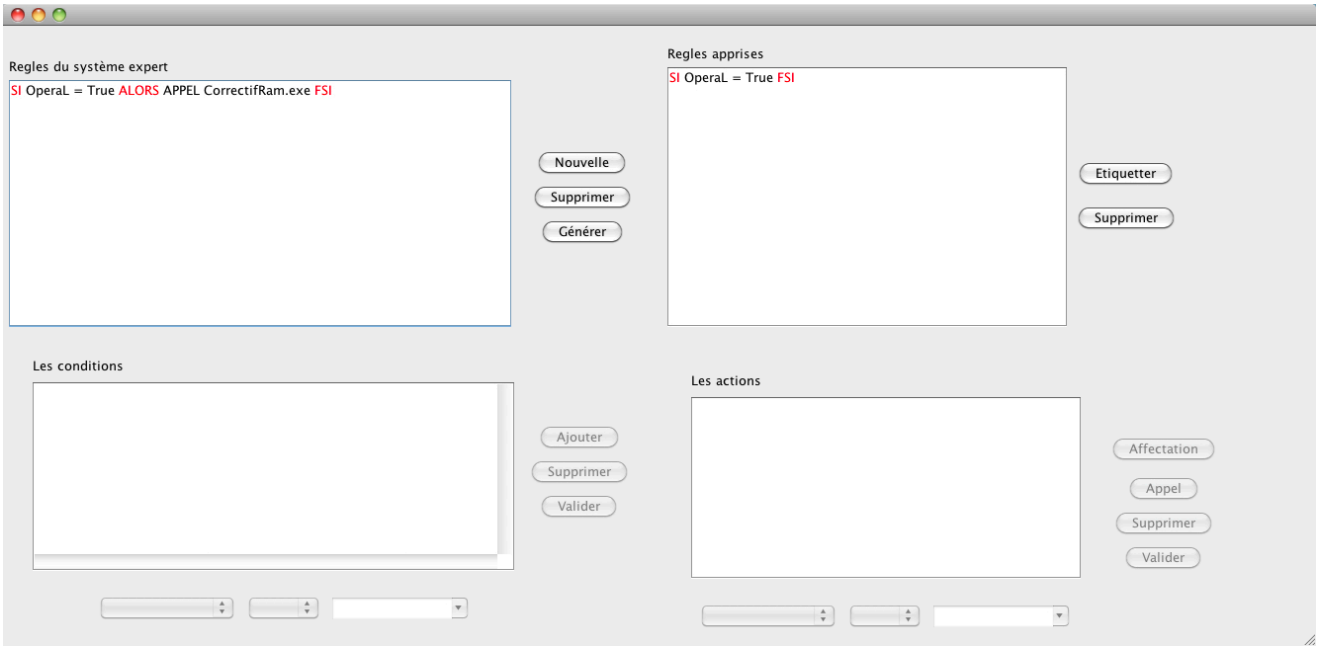

FIGURE 1 INTERFACE GRAPHIQUE ADMINISTRATEUR

La figure ci-dessus représente une vue de l'interface utilisateur, peu modifiée en profondeur, on y trouve 4 parties :

- Une fenêtre permettant de visualiser les règles du système expert (en haut à gauche). Dans cet exemple, si OperaL est à vrai, alors on applique le correctif CorrectifRam.exe.
- Une fenêtre ayant pour but d'afficher les règles apprises (en haut à droite). Dans ce cas, seulement une règle est présente : Si OperaL est vrai alors il y'a eu plantage.
- Une fenêtre permettant de créer ou modifier des conditions après avoir sélectionné une des règles apprises (en bas à gauche).
- Une fenêtre permettant de créer ou modifier des actions. Une fois les actions et conditions validées, la règle est ajoutée au système et apparait dans la fenêtre en haut à gauche.

La conception de l'interface utilisateur reste donc relativement identique à celle de l'année dernière. Le diagramme de classe ci-dessous, fait ressortir les éléments physiques constituant l'interface :

- Une classe FenetreRegleSE modélisant la fenêtre en haut à gauche permettant de visualiser dans un premier temps les règles déjà utilisées par le système expert et de visualiser les règles ajoutées. Si l'on regarde les règles apprises à partir des arbres de décision elles sont toutes mutuellement exclusives, l'ordre de ces règles n'a donc

strictement aucune importance. Par contre si l'on effectue des modifications manuelles sur ces règles, l'ordre peut devenir important, c'est pourquoi la fenêtre permet de modifier à la souris leur ordre.

- Une classe FenetreRegleApp modélisant la fenêtre en haut à droite, celle permettant de visualiser l'ensemble des règles venant d'être apprises.
- Une classe FenetreCondition modélisant la fenêtre en bas à gauche et permettant de saisir de nouvelles conditions.
- Une classe FenetreAction modélisant la fenêtre en bas à droite et permettant de saisir de nouvelles conditions.

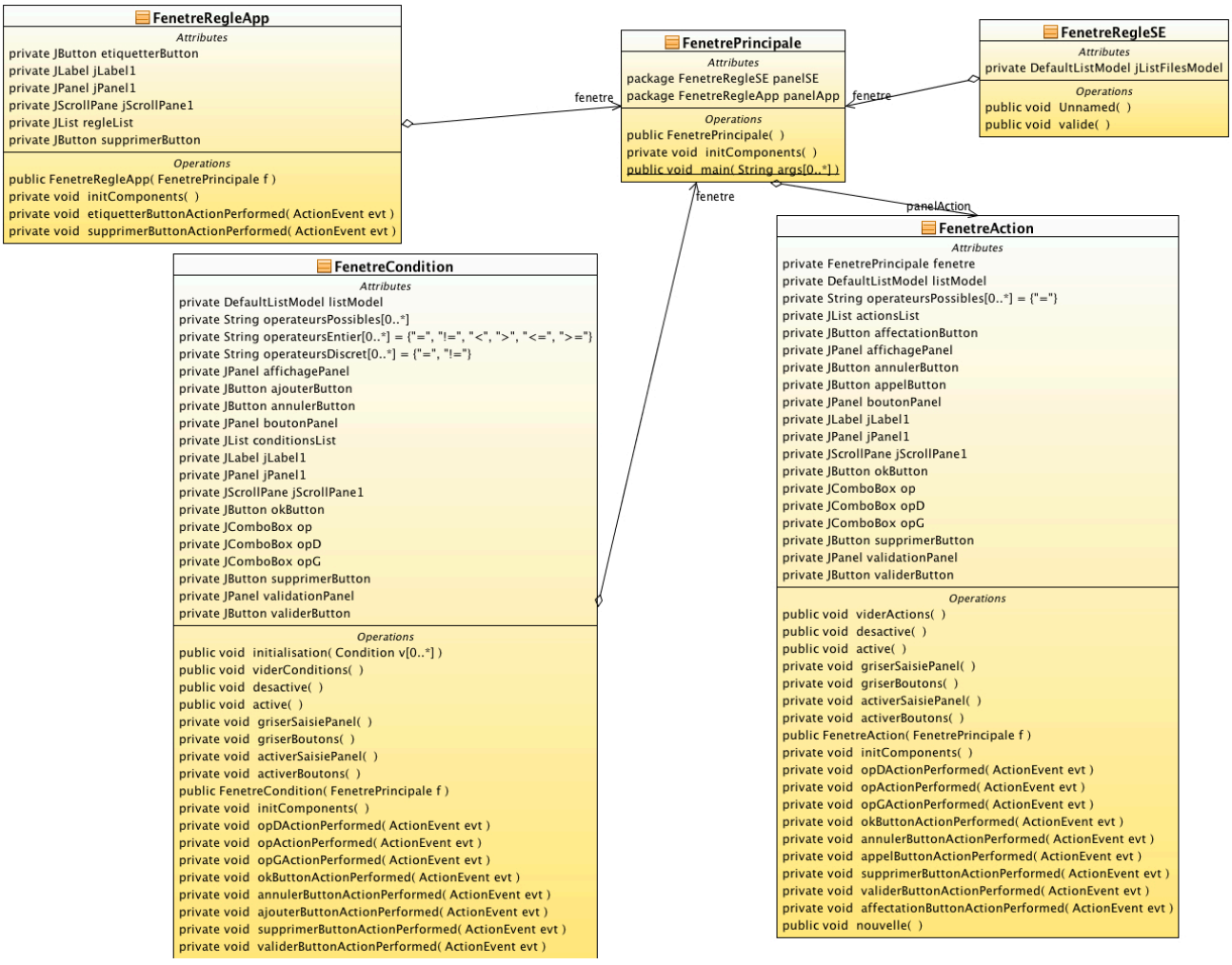

DIAGRAMME 21 DIAGRAMME DE CLASSES DU SIMULATEUR

Si la conception reste proche de celle de l'année dernière, le code risque d'être modifié : en effet cette année une nouvelle conception de la base de faits et de règles a été faite. On aurait pu penser que pour la partie interface Administrateur seule la partie entrées-sorties allait être modifiée, mais ce n'est pas le cas. En effet, dans chacune des classes ci-dessus on trouve des attributs liés à la base de faits qui permettent de stocker les informations :

- Dans la fenêtre FenetreRegleSE on trouve deux attributs : lesRegles de type Vector<Regle> et r de type RegleSE. Une règle contient simplement une partie condition alors que la regleSE contient à la fois une partie condition et action. Sur ces attributs deux méthodes sont utilisées : ajoutActions(Vector<Action>) et ajoutConditions(Vector<Condition>).
- Dans la fenêtre FenetreRegleApp, il n'y a pas d'attributs propres, par contre cette classe utilise les attributs statics du programme principal (classe Learning1). En particulier Learning1.machine de type MachineARegle est utilisée pour aller chercher les règles apprises et les afficher. Les méthodes de la classe MachineARegle utilisées sont getRegles qui retourne un vector<Regle> et lesRegles() qui retourne l'ensemble des règles sous forme de chaines de caractères.
- Dans la fenêtre FenetreAction tous les éléments constituant une action apparaissent en attribut : OpAttribut operandeG, Operande operandeD et Operateur opérateur. Ce sont les constructeurs de ces classes qui sont utilisés et les constructeurs de la classe Action :

  Action c = new Affectation(String,String)

  Action c = new Appel(String)

Si l'on fait disparaître la partie Affectation cela signifie qu'il faut modifier la fenêtre action.

- Dans la fenêtre FenetreCondition on retrouve aussi tous les attributs permettant de stocker une condition : OpAttribut operandeG, Operande operandeD, Operateur operateur. Là aussi ce sont les constructeurs de ces classes qui sont utilisés et le constructeur de la classe Condition :

  Condition c = new Condition(this.operandeG, this.operateur, this.operandeD)

4.4. Génération

Pour la génération du fichier de règles en XML, il suffira d'appeler la méthode toXML de la base de règles du serveur, qui sera chargée de récupérer chaque règle sous format XML à l'aide également de la méthode toXML de chaque objet Regle.

5. Simulateur

5.1. Fonctionnement Général

5.1.1. Description globale

Le simulateur est une partie importante du projet puisqu'il va nous permettre de générer des exemples d'exécution du système expert en grande quantité et en peu de temps, et ainsi de tester l'algorithme utilisé pour l'apprentissage.

Il est composé de différents modules :

- Le module Etat qui représente l'état du téléphone
- Le Contrôleur qui lance les différentes applications et applique les effets à l'état
- Le système expert
- Le système de reporting
- L'horloge qui représente l'heure du téléphone

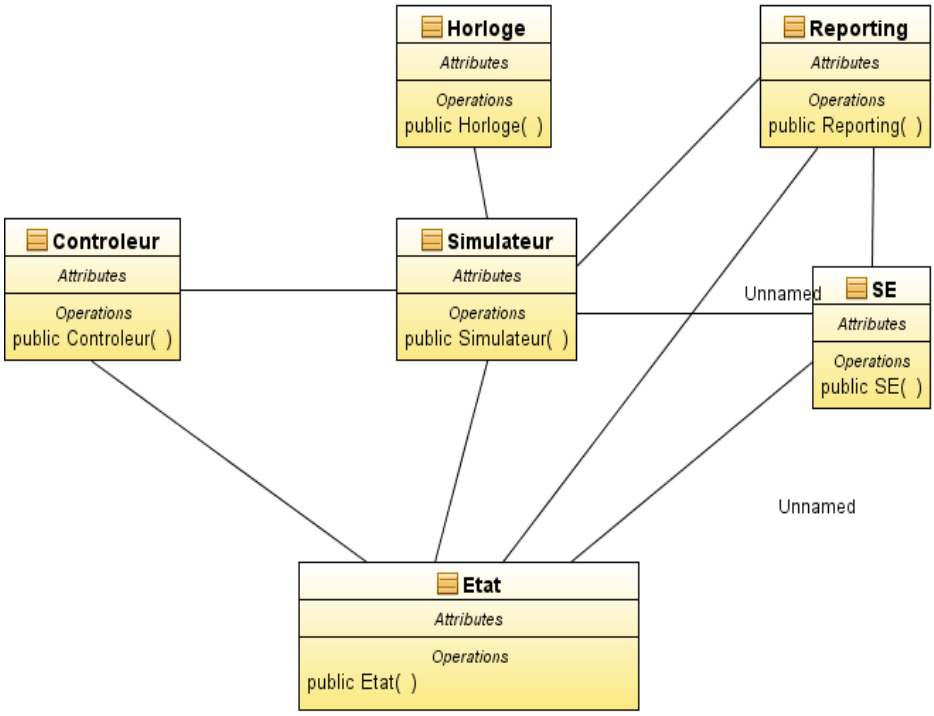

DIAGRAMME 22 DIAGRAMME DE CLASSE GLOBAL DU SIMULATEUR

5.1.2. Description des entrées

Le simulateur prend en entrée plusieurs fichiers au format XML :

- Un fichier application qui décrit les différentes applications présentes sur le téléphone :

```
<applications>
	<application id="1" nom="AppliA" >
		<frequence min="180" max="240" />
		<duree min="10" max="20" />
		<effets>
			<effet type="ajout_temporaire">
				<target nom="mémoireVive" id="1" />
				<value valeur="10" />
			</effet>
			<effet type="ajout_périodique">
				<target nom="batterie" id="2" />
				<interval valeur = "10" />
				<value valeur="-1" />
			</effet>
		<effets>
	</application>

	<application id="2" nom="AppliB" >
		<frequence min="120" max="120" />
		<duree min="10" max="60" />
		<effets>
			<effet type="ajout_périodique">
				<target nom="batterie" id="2" />
```

```
                        <interval valeur = "10" />
                        <value valeur="-3" />
                </effet>
        <effets>
    </application>

    <application id="3" nom="AppliC" >
        <frequence minj="1" maxj="3" />
        <duree min="120" max="360" />
        <effets>
            <effet type="ajout_permanent">
                <target nom= "memoirePhysique" id="5"/>
                <value valeur = "10" />
            </effet>
            <effet type="ajout_périodique">
                <target nom="batterie" id="2" />
                <interval valeur = "15" />
                <value valeur="-1" />
            </effet>
        <effets>
    </application>
</applications>
```

Les applications possèdent un nom, un id, un attribut fréquence qui représente sa fréquence de lancement, un attribut durée, ainsi qu'un attribut effet. On distingue trois types d'effets, les effets temporaires qui vont être appliqués au lancement et retirés à son arrêt, les effets permanents, et les effets périodiques qui s'appliquent régulièrement.

- Un fichier des attributs du téléphone semblable à celui utilisé par l'interface administrateur :

```
<attributs>
    <attribut id="1" nom="mémoireVive" type="numeric" min="0" max="1000"
/>
    <attribut id="2" nom="batterie" type="numeric" min="0" max="100" />
    <attribut id="3" nom="plantage" type="enum" >
        <valeur>Oui<valeur/>
        <valeur>Non<valeur/>
    <attribut/>
    <attribut id="5" nom="mémoirePhysique" type="numeric"  min="1000"
max="8000" />
<attributs>
```

- Un fichier complémentaire sur les attributs, contenant des paramètres utilisés spécifiquement par le simulateur :

```
<attributs>
    <attribut id="1" nom="mémoireVive"  default="200" reset="true" />
    <attribut id="2" nom="batterie" default="100"  reset="false" />
    <attribut id="3" nom="plantage" default="non" reset="true" />
    <attribut id="4" nom="typeplantage" default="AppliCrash"
reset="true"/>
    <attribut id="5" nom="mémoirePhysique" default="1000" reset="false"/>
<attributs>
```

Le paramètre reset indique si le paramètre est à remettre à sa valeur d'origine (défault) lors d'un reboot du téléphone.

- Un fichier décrivant les différentes règles de plantage du téléphone

```
<rules>
```

```
	<rule ID="1" >
		<conditions>
			<condition type="between">
				<subject type="Attribut" id="1" />
				<Integer value="900" >
				<Integer value="1000" >
			</condition>
			<condition value="equals">
				<subject type="Appli" id="3" />
				<String value="lancer" />
			</condition>
		</conditions>
	</rule>
	<rule ID="2" >
		<conditions>
			<condition type="between">
				<subject type="Attribut" id="1" />
				<min value="900" >
				<max value="1000" >
			</condition>
			<condition value="equals">
				<subject type="Appli" id="1" />
				<String value="lancer" />
			</condition>
		</conditions>
	</rule>
	<rule ID="3" >
		<conditions>
			<condition value="equals">
				<subject type="Appli" id="1" />
				<String value="lancer" />
			</condition>
			<condition value="equals">
				<subject type="Appli" id="2" />
				<String value="lancer" />
			</condition>
		</conditions>
	</rule>
	<rule ID="4" >
		<conditions>
			<condition type="between">
				<subject type="Attribut" id="2" />
				<min value="5" />
				<max value="10" />
			</condition>
			<condition value="equals">
				<subject type="Appli" id="1" />
				<String value="lancer" />
			</condition>
		</conditions>
	</rule>
</rules>
```

5.1.3. Description des sorties

Les sorties du simulateur sont en fait les rapports générés en boucle par le système de reporting, semblables aux rapports générés pour les Smartphones (les conceptions des deux systèmes de reporting étant semblables).

De plus l'ensemble des actions effectuées sur l'état par le simulateur est enregistré dans un fichier log, ce qui nous permet de surveiller le déroulement de la simulation.

5.1.4. Déroulement de l'exécution

Le simulateur initialise l'état à l'aide des fichiers pris en entrée, et lance le contrôleur et le système de reporting qui s'exécutent en boucle (utilisation de threads). Lorsqu'un plantage est détecté on simule un reboot du téléphone puis on reprend l'exécution. Lors du reboot l'état du téléphone se réinitialise en tenant compte des effets réversibles ou non sur les attributs (définis dans le fichier complémentaire sur les attributs avec le paramètre *reset*). Ainsi par exemple la batterie ne sera pas réinitialisée à sa valeur d'origine (paramètre *default*) mais conservera sa valeur actuelle.

5.2. Etat

L'état représente l'état interne du mobile simulé. Il est composé de 2 parties.

La première est la liste des attributs qui peuvent être de type numérique ou énumérés. Ils possèdent un identifiant numérique unique ainsi qu'un nom littéral. Les attributs numériques sont bornés. Les attributs énumérés possèdent la liste des valeurs possibles ainsi que la valeur courante. Ces informations sont chargées à partir de fichier attributs.xml qui est aussi utilisé par l'interface administration pour la construction des règles. Le fichier attributSimu.xml contient les propriétés des attributs qui sont propre au simulateur. Premièrement la valeur par défaut qui est utilisé à l'initialisation de l'attribut et deuxièmement la propriété booléen reset qui indique si lors d'un reboot de mobile simulé l'attribut dois être remise à sa valeur par défaut. Exemple la mémoire vive du téléphone est remise à zéro alors que le taux de batterie reste le même qu'avant le reboot.

La deuxième partie contient les états courants de l'ensemble des applications du mobile. Chaque application possède comme les attributs, un identifiant et un nom littéral. Elles permettent de savoir si l'application est lancée. Si c'est le cas un compteur de duré indique pendant combien de temps l'application doit rester allumée, dans le cas contraire un autre compteur indique le temps restant avant la prochaine activation. Ces deux compteurs sont initialisés par des valeurs aléatoires compris entre un max et un min. Les applications possèdent également une liste d'effets. Ils agissent sur les attributs. L'effet Setter change la valeur d'un attribut énuméré au lancement de l'application. Pour les attributs numériques, on peut ajouter une valeur (qui peut être négative). L'ajout peut se faire de 3 manières : un ajout temporaire, ou la valeur ajoutée est changée au lancement et retirée à l'arrêt de l'application. Un ajout permanant, ou la valeur est ajoutée mais pas retiré. Un ajout périodique, qui ajoute régulièrement une valeur de façon permanente. Un exemple serait la diminution progressive de

la batterie. Les effets périodiques sont stockés dans une liste différente puisqu'ils sont appelés plus souvent que les autres effets.

La fonction reboot simule un redémarrage du Smartphone en remettant les valeurs par défaut des attributs appropriés et en fermant toutes les applications.

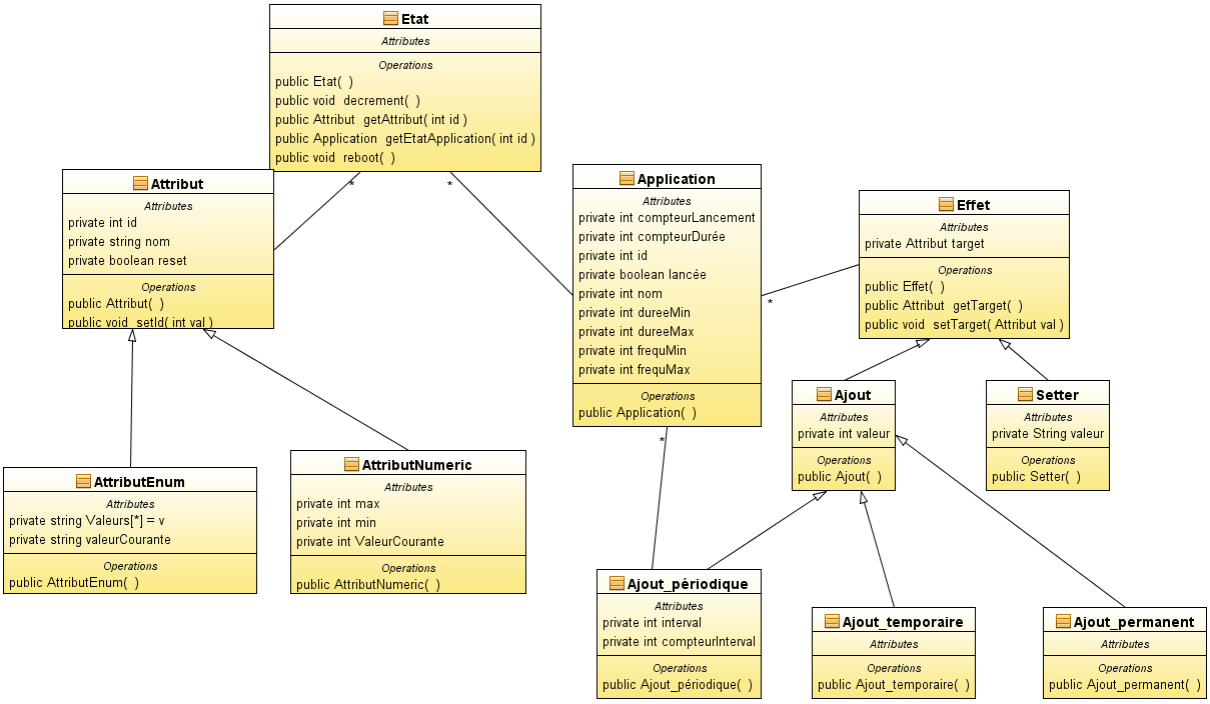

DIAGRAMME 23 DIAGRAMME DE CLASSE DE L'ETAT INTERNE DU MOBILE SIMULE

5.3. Contrôleur

Le contrôleur simule la dynamique du Smartphone et l'apparition de plantage. Les règles de plantages sont chargées à partir du fichier regleTel.xml et sont ensuite stockées dans une structure identique à celle de la base de règle. La seule action de cette base de règles est le redémarrage du Smartphone. Le contrôleur anime les applications. Il demande à l'état de décrémenter leurs compteurs. Si un compteur d'activation arrive à zéro, l'application correspondante est lancée, ses effets temporaires et permanents sont appliqués, les compteurs des effets périodiques sont initialisés. Le contrôleur vérifie si des règles de plantage sont applicables. Si c'est le cas, l'état simule un reboot, sinon on passe à la décrémentation du compteur de durée. Si l'application possède des effets périodiques, on les applique en temps voulu, suivi systématiquement d'une vérification des règles de plantage.

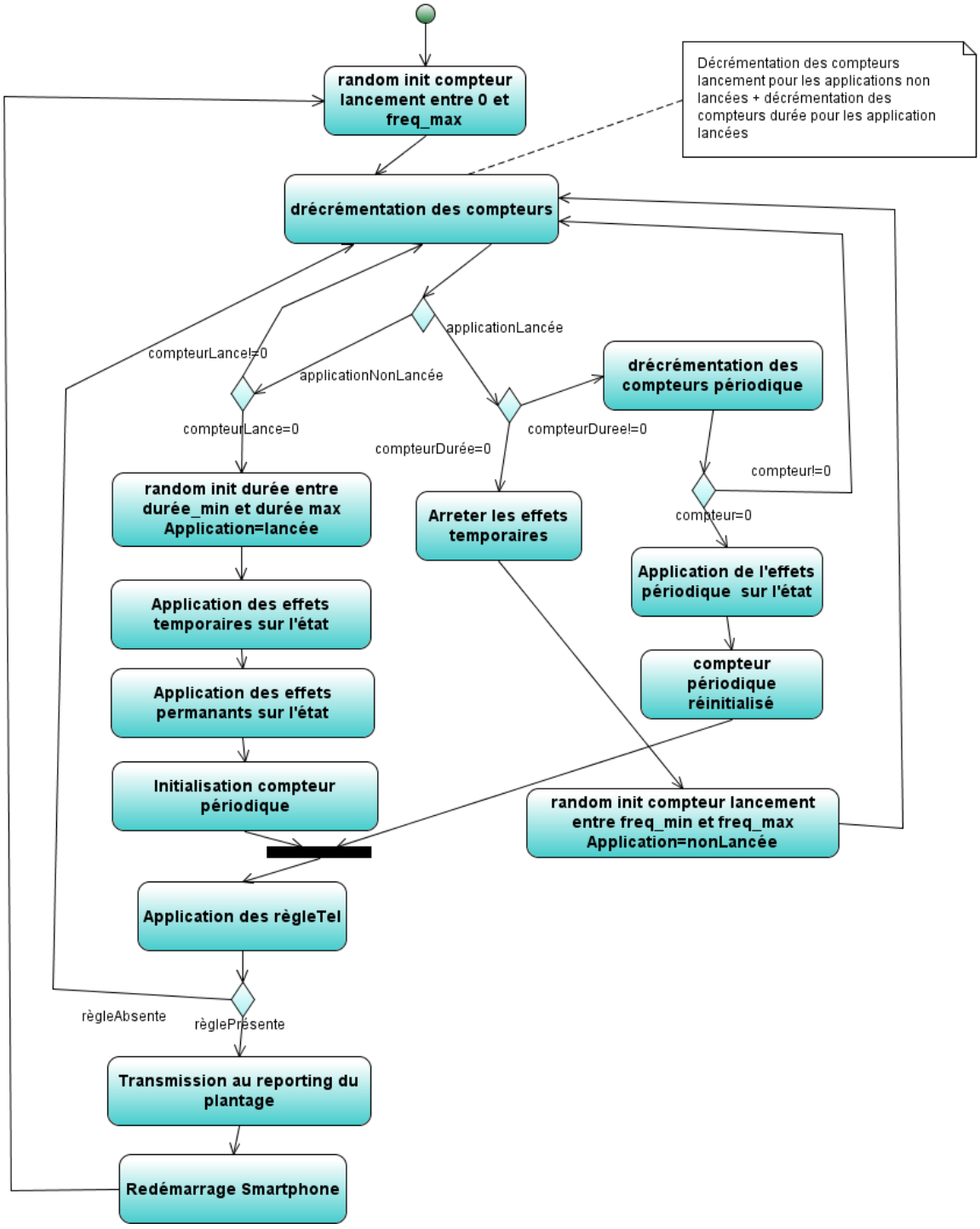

DIAGRAMME 24 DIAGRAMME D'ACTIVITE DU CONTROLEUR

5.4. Système Expert et Reporting

Le système expert et le module de reporting du simulateur et du mobile sont semblables (voir Partie Mobile). Les seules différences résident du fait que le système de reporting du simulateur est adapté pour récupérer les informations sur l'état (alors que le

reporting de la partie mobile les récupère sur le téléphone). De même pour le système expert qui applique les règles à l'état du simulateur au lieu du téléphone. De plus le reporting recevra en cas de bug un signal de la part du contrôleur, comme l'indique le diagramme suivant.

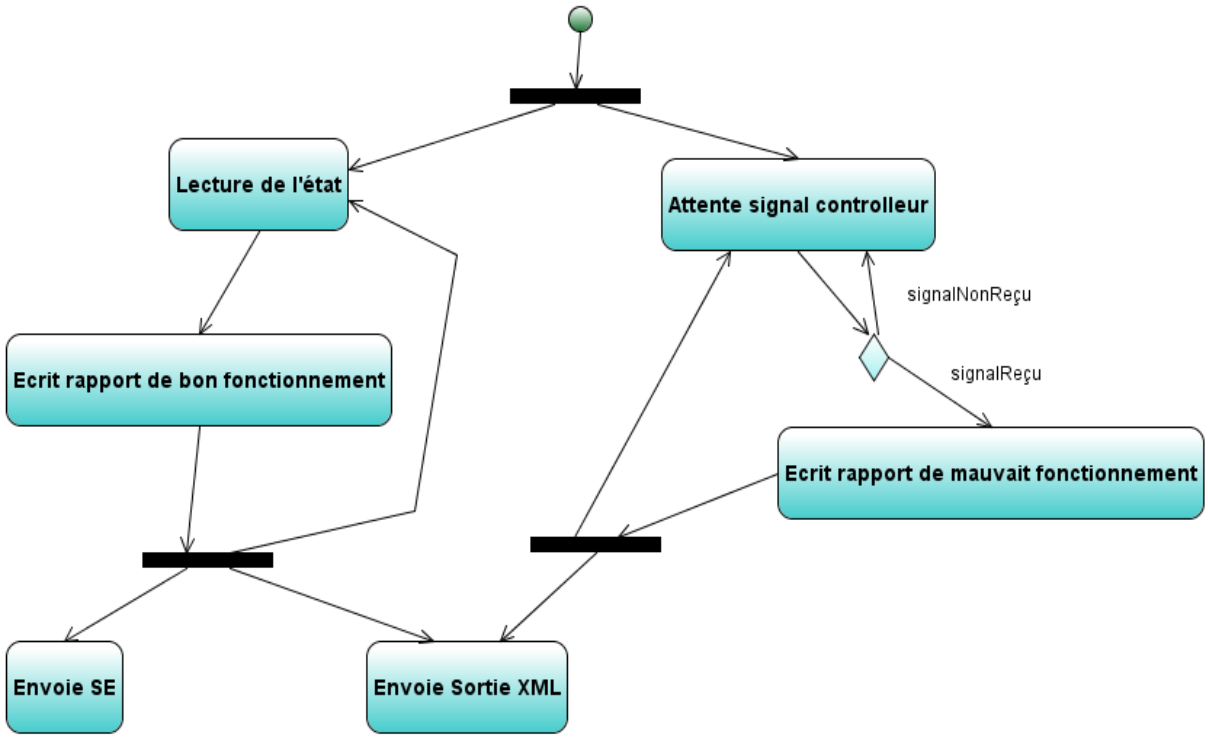

DIAGRAMME 25 DIAGRAMME D'ACTIVITE DE MODULE DE REPORTING ET DU SYSTEME EXPERT SIMULE

6. Conclusion

Nous avons vu la modélisation des différents modules de la partie mobile qui sont le système expert, le système de reporting et le suivi utilisateur Android, ainsi que la modélisation de la partie serveur sur laquelle se trouve l'apprentissage des règles ainsi que l'interface administrateur.

Les rapports de bon et mauvais fonctionnement sont écrits en XML pour le système Android. Pour le système de l'iOS, les rapports de bon fonctionnement sont écrits en XML tandis que les rapports d'erreurs seront créés en XML à partir des crash-log générés par l'iOS. La communication entre les différents modules passe donc par un même langage pour une avoir une lecture et écriture de fichiers assez générale, et obtenir un système globale uniformisé.

Nous allons maintenant passer à la phase de développement tout en effectuant les tests en parallèle. Certains modules peuvent être développés indépendamment ce qui sera un avantage concernant le fonctionnement en sous équipes dans notre projet. De plus le développement du simulateur en Java nous permettra d'obtenir un code réutilisable pour le développement du système Android également dans ce même langage.

7. Bibliographie

1. **iPuP.** *Programmez pour iPhone, iPod Touch, iPad avec iOS4.* s.l. : Pearson, 2010.

2. **Appcelerator, Inc.** Appcelerator. [En ligne] [Citation : 21 10 2010.] http://www.appcelerator.com/.

3. Xsysinfo - Display bar graphs of system load. *Linux Software Directory.* [En ligne] [Citation : 27 Septembre 2010.] http://linux.maruhn.com/sec/xsysinfo.html.

4. **WITTEN, Ian H. et FRANK, Eibe.** *Data Mining.* s.l. : Morgan Kaufmann, 2005.

5. **McEntire, Norman.** How to use iPhone with Unix System and Library Calls : A Tutorial for Software Developers. *How to use iPhone with Unix System and Library Calls.* [En ligne] 18 Janvier 2009. [Citation : 27 09 2010.] http://www.servin.com/iphone/iPhone-Unix-System-Calls.html.

6. **Guy, Romain.** Painless threading. *Android Developers.* [En ligne] 2009. http://android-developers.blogspot.com/2009/05/painless-threading.html.

7. **Collins, Charlie.** Android Application and AsyncTask basics. [En ligne] 2010. http://www.screaming-penguin.com/node/7746.

8. **Apple Inc.** iOS Reference Library. [En ligne] 2010. http://developer.apple.com/library/ios/navigation/.

9. **Google.** Android SDK. [En ligne] 2010. http://developer.android.com/sdk/index.html.