Licence WEB
IUT de Clermont-Ferrand
Année scolaire 2006-2007

Aurélien JOAHNY
Benoît MARIUS

Validation de services web

JAVA, AXIS, XML

Rapport de projet

Tuteur : M SALVA

Sommaire :

Introduction

Qu'est ce qu'un Web Service ?

Les Web Services, ou Services Web, implémentent de la logique métier rendue consommable, par l'utilisation de standards, majoritairement TCP/IP, URI/URN/URL, MIME, HTTP/SMTP/..., SOAP, SSL/TLS, ... pour le transport, puis XML pour le contenu, ce qui permet à n'importe quelle technologie utilisant ces standards de pouvoir l'exploiter, facilitant ainsi l'interopérabilité des applications. On dit qu'on consomme le Web Service.
La création de Web Services se justifie par l'architecture orientée service, c'est à dire la volonté de rendre accessible un service qui implémente une logique métier cachée à des utilisateurs.
Concrètement un Web Service permet à un objet d'invoquer des méthodes d'objets physiquement situés sur une autre machine, généralement en utilisant internet.
Dans le cadre de contrats d'échanges de données en Business to Business , ou d'entreprise à entreprise, comme en Business to Consumer, ou d'entreprise à client/utilisateur. Un autre intérêt pour lequel des services Web sont employés est le fait qu'ils se fondent sur le protocole HTTP, qui utilise le port 80 par défaut. Beaucoup d'entreprises se sont protégées en employant des firewalls qui filtrent et bloquent beaucoup de trafic d'Internet pour des raisons de sécurité. Dans ce milieu, presque tous les ports sont fermés au trafic entrant et sortant. Et les administrateurs de ces firewalls ne sont pas désireux de les ouvrir. Le port 80, cependant, est toujours ouvert parce qu'il est employé par le protocole HTTP utilisé par les navigateurs Web.

Les technologies utilisées :

Afin d'utiliser un Web Service, on doit pouvoir communiquer avec lui. Dans ce but certains protocoles existent, tous basés sur le XML :

XML :

XML ou Extensible Markup Language, langage de balisage extensible, est un langage informatique de balisage générique. Le W3C recommande XML pour exprimer des langages de balisages spécifiques, comme le XHTML le SVG et le XSLT. Son objectif initial est de faciliter l'échange automatisé de contenus entre systèmes d'informations hétérogènes à des fins d'interopérabilité, notamment sur Internet. XML est un sous-ensemble de SGML dont il retient plusieurs principes comme : La structure d'un document XML est définissable et validable par un schéma. Un document XML est entièrement transformable en un autre document XML. Cette syntaxe est reconnaissable par son usage des chevrons, < >, et s'applique à de plus en plus de contenus.

WSDL :

La description WSDL d'un Web Service renseigne le consommateur sur les méthodes dont il dispose, leurs paramètres et type de retour. C'est une description basée sur le XML qui indique comment communiquer pour utiliser le service, le protocole de communication, et le format de messages requis pour communiquer avec ce service. Les opérations possibles et messages sont décrits de façon abstraite mais cette description renvoie à des protocoles réseaux et formats de messages concrets. C'est le contrat de services.

SOAP :

(Simple Object Access Protocol ou Service Oriented Architecture Protocol) SOAP est le protocole de dialogue entre le Web Service et le consommateur. Sa normalisation permet l'interconnexion. Cependant il existe plusieurs modèles de message de SOAP : le modèle RPC et le modèle Document/literal. Pour notre projet nous utiliseront le modèle Document/literal wrapped, celui-ci étant une évolution du Document/literal, modèle aujourd'hui le plus couramment utilisé sur Internet.

AXIS :

Axis est un projet de la Apache Software Foundation. C'est un package Java qui fournit :

Un environnement pouvant soit fonctionner comme un serveur SOAP indépendant soit comme un plug-in de moteurs de servlet, en particulier Tomcat.

Une API pour développer des services web SOAP RPC ou à base de messages SOAP.

Le support de différentes couches de transport : HTTP, FTP...

La sérialisation/désérialisation automatique d'objets Java dans des messages SOAP.

Des outils pour créer automatiquement les WSDL correspondant à des classes Java ou inversement pour créer les classes Java sur la base d'un WSDL.

Des outils pour déployer, tester et monitorer des web-services.

Le projet s'appuie sur un Web Service en JAVA qui utilise AXIS pour comuniquer avec d'autres Web Service, mais aussi avec ses clients.

SAX :

SAX est une API JAVA de parsing XML utilisée pour parser le WSDL dans le projet. Elle propose un accès au contenu dans l'ordre des balises rencontré dans le fichier XML.

DOM :

DOM, ou Document Object Model, est une API JAVA de parsing et d'édition XML, utilisée pour génerer des schéma XML dans le projet. DOM propose un accès au contenu en fonction de la structure du XML. Il a donc besoin de charger intégralement le fichier XML en mémoire avant de pouvoir y naviguer, ce qui le rend inadapté pour des grosses structures XML.

JAXB :

API de manipulation du protocole SOAP.

Problématique :

Les Web Services se développent très rapidement un peu partout. Ils présentent de nombreux avantages, principalement de pouvoir consommer des services sans avoir à les développer. Cela est possible grâce à la description WSDL du Web Service qui renseigne sur les méthodes du service. Cependant l'utilisateur n'a pas la main sur le service, et est alors impuissant vis-à-vis des éventuels bugs de celui-ci. De plus beaucoup de Web Services ne respectent pas leur description WSDL, seul repère pour le consommateur de service.

Face à ce constat, notre projet consiste à mettre au point une application capable de vérifier le fonctionnement d'un Web Service :
A partir de l'adresse d'un Web Service, l'application doit pouvoir en récupérer la description WSDL. Grâce à la description WSDL, l'application doit construire un ensemble de scénarios tests. Par la suite ces scénarios doivent être exécutés par Internet sur le Web Service, et leurs résultats, une fois analysés, permettront de valider ou non le Web Service.

Réflexion

Étude préalable :

1) ***Étude de l'existant :***

- *Démarche d'un développeur de Web-Service :*

1. *Développement du Web-Service.*

2. *Génération automatique de la description WSDL.*

3. *Test du Web-Service :*

    - *Jeux d'essais du développeur ou du testeur.*
    - *Applicatif de test/performance très onéreux.*
    - *Programmation d'un applicatif ciblé pour le Web-Service.*

2) ***Critique de l'existant :***

- *Web-Service testé, mais aucune vérification de la cohérence du Web-Service avec sa description WSDL :*

1. *La gestion des erreurs est elle correcte ?*

    - *Toutes les exceptions susceptibles de se produire sont-elles déclarées dans la description WSDL ?*
    - *Le Web-Service plante t-il ?*

2. *Les méthodes du Web-Service sont-elles assez robustes :*

    - *Division par zéro.*
    - *Pointeur nul.*
    - *Valeurs extrêmes pour chaque type simple de variables.*
    - *Valeurs en-dehors des bornes des types de variables.*
    - *Répétition d'une multitude de requêtes.*

3) ***Objectifs :***

- *Vérifier la cohérence entre la description WSDL du Web-Service et le Web-Service.*
- *Contrôler la gestion des erreurs du Web-Service.*
- *Tester la robustesse du Web-Service.*

4) ***Besoins :***

- *L'application doit pouvoir rendre un rapport sur le Web-Service au niveau de :*

  1. *Sa cohérence avec sa description WSDL.*
  2. *Sa gestion des erreurs.*
  3. *Sa robustesse.*

- *Outil pour les développeurs et pour les consommateurs de Web-Services.*

- *Outil automatique: l'utilisateur doit seulement lui fournir l'url du Web-Service.*

- *Utilisation avancée : l'utilisateur doit pouvoir choisir ses propres paramètres à envoyer aux méthodes du Web-Service.*

5) ***Réalisation :***

- *Développement d'un Web Service JAVA et Axis. Choix du Web Service car dans le cas d'une utilisation simple de l'application, l'utilisateur n'as que très peu d'interaction avec l'application, mais l'application a, elle, besoin d'effectuer un grand nombre de tests auprès du Web Service à tester, ce qui nécessite un besoin important en therme de coût réseau. En règle générale les serveurs de Service Web on une bien meilleur connexion que l'utilisateur qui effectuera les tests. L'application que nous allons réaliser, et le Web Serviceà tester, pourront donc communiquer beaucoup plus efficacement dans ce cas.*

- *Utilisation du protocole SOAP 1.2 Document/Literal wrapped.*

- *Utilisation type de l'application pour l'utilisateur :*

  1. *L'utilisateur renseigne l'application sur l'url du Web-Service.*
  2. *L'utilisateur lance la validation.*
  3. *L'application renseigne l'utilisateur sur la validité du Web-Service, et lui présente un rapport en cas de non-validation.*

- *Utilisation avancée de l'application pour l'utilisateur :*

  1. *L'utilisateur renseigne l'application sur l'url du Web-Service.*
  2. *L'utilisateur spécifie qu'il souhaite utiliser l'application de manière avancée.*
  3. *L'utilisateur lance la récupération de la description WSDL.*
  4. *L'utilisateur choisit la méthode à tester du Web-Service.*
  5. *L'utilisateur renseigne les paramètres à utiliser.*
  6. *L'utilisateur lance la validation de la méthode choisie.*
  7. *L'application renseigne l'utilisateur sur la validité de la méthode du Web-Service, et lui présente un rapport en cas de non-validation.*

- *Utilisation type de l'application :*

  1. *L'utilisateur renseigne l'application sur l'url du Web-Service.*
  2. *L'utilisateur lance la validation.*
  3. *L'application récupère la description WSDL du Web-Service.*
  4. *L'application génère un jeu d'essais pour le Web-Service.*
  5. *L'application test son jeu d'essais par des requêtes au Web-Service.*
  6. *L'application reçoit et interprète les résultats.*
  7. *L'application renseigne l'utilisateur sur la validité du Web-Service, et lui présente un rapport en cas de non-validation.*

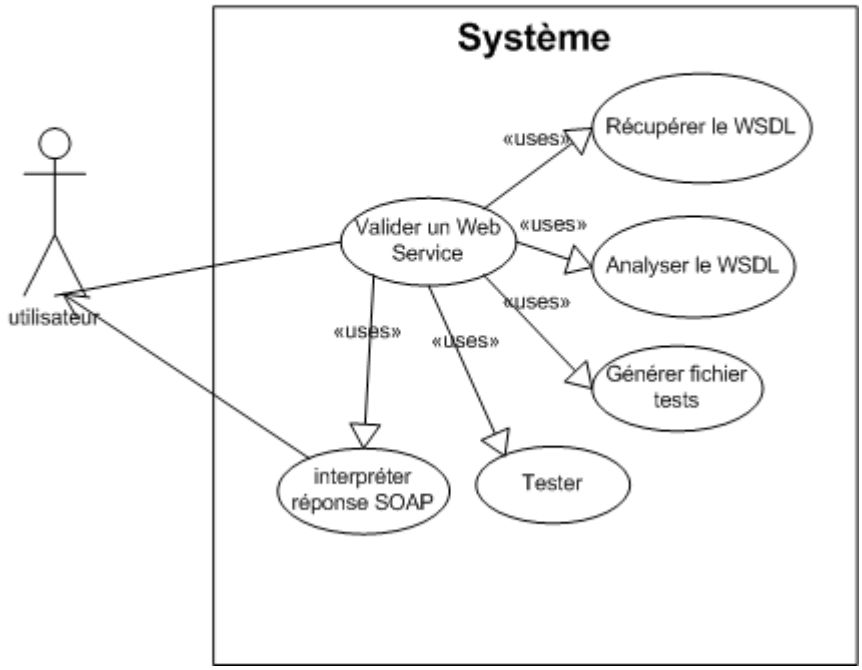
- *Utilisation avancée de l'application :*

1. *L'utilisateur renseigne l'application sur l'url du Web-Service.*
2. *L'utilisateur spécifie qu'il souhaite utiliser l'application de manière avancée.*
3. *L'utilisateur lance la récupération de la description WSDL.*
4. *L'application récupère la description WSDL du Web-Service.*
5. *L'utilisateur choisit la méthode à tester du Web-Service.*
6. *L'utilisateur renseigne les paramètres à utiliser.*
7. *L'utilisateur lance la validation de la méthode choisie.*
8. *L'application test le jeu d'essais de l'utilisateur par une ou des requêtes au Web-Service.*
9. *L'application reçoit et interprète les résultats.*
10. *L'application renseigne l'utilisateur sur la validité de la méthode du Web-Service avec les paramètres que l'utilisateur a saisis, et lui présente un rapport en cas de non-validation.*

Analyse

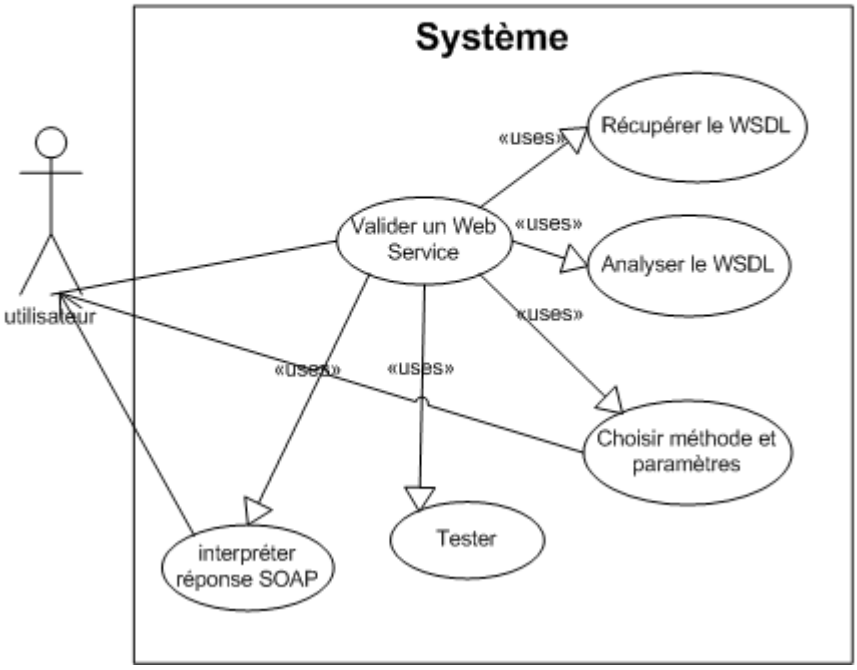
valider un web service - utilisation type

cas d'utilisation



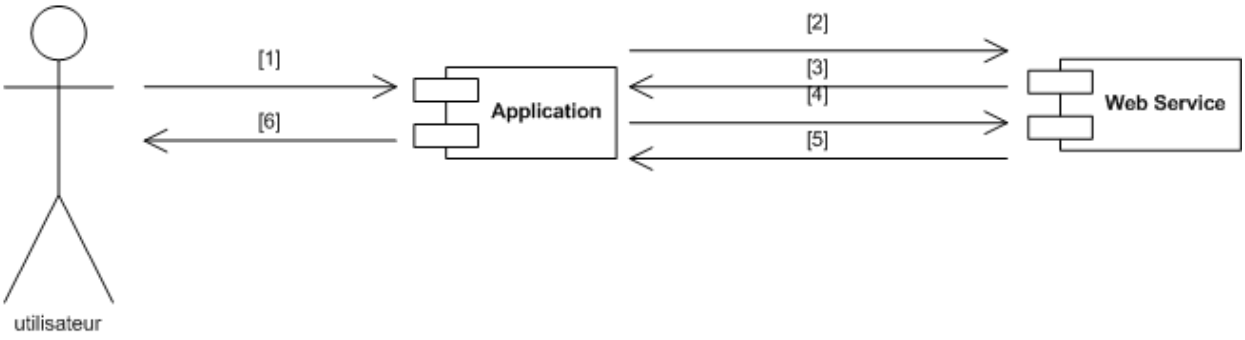
valider un web service - utilisation avancée

cas d'utilisation



utilisation type de l'application

diagramme d'interaction



1) L'utilisateur renseigne l'url du web service et lance la validation.

2) L'application demande la description WSDL au web service.

3) Le web service envoie la description WSDL à l'application.

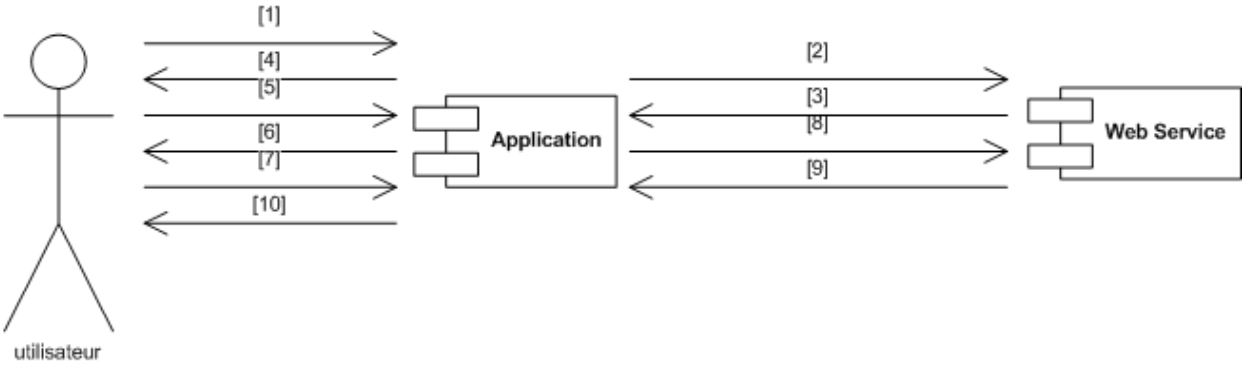
4) L'application test le jeu d'essai, qu'elle a généré grace à la description WSDL, par des requêtes au web service.

5) Le web service répond en SOAP aux requêtes.

6) L'application communique à l'utilisateur la validité du web service par rapport à sa description WSDL, et lui présente un rapport en cas de non validation.

utilisation avancée de l'application

diagramme d'interaction



1) L'utilisateur renseigne l'url du web service, il spécifie l'utilisation de l'application en mode avancé et lance la récupération de la description WSDL.

2) L'application demande la description WSDL au web service.

3) Le web service envoie la description WSDL à l'application.

4) L'application propose à l'utilisateur le choix de la méthode à tester.

5) L'utilisateur choisit la méthode à tester.

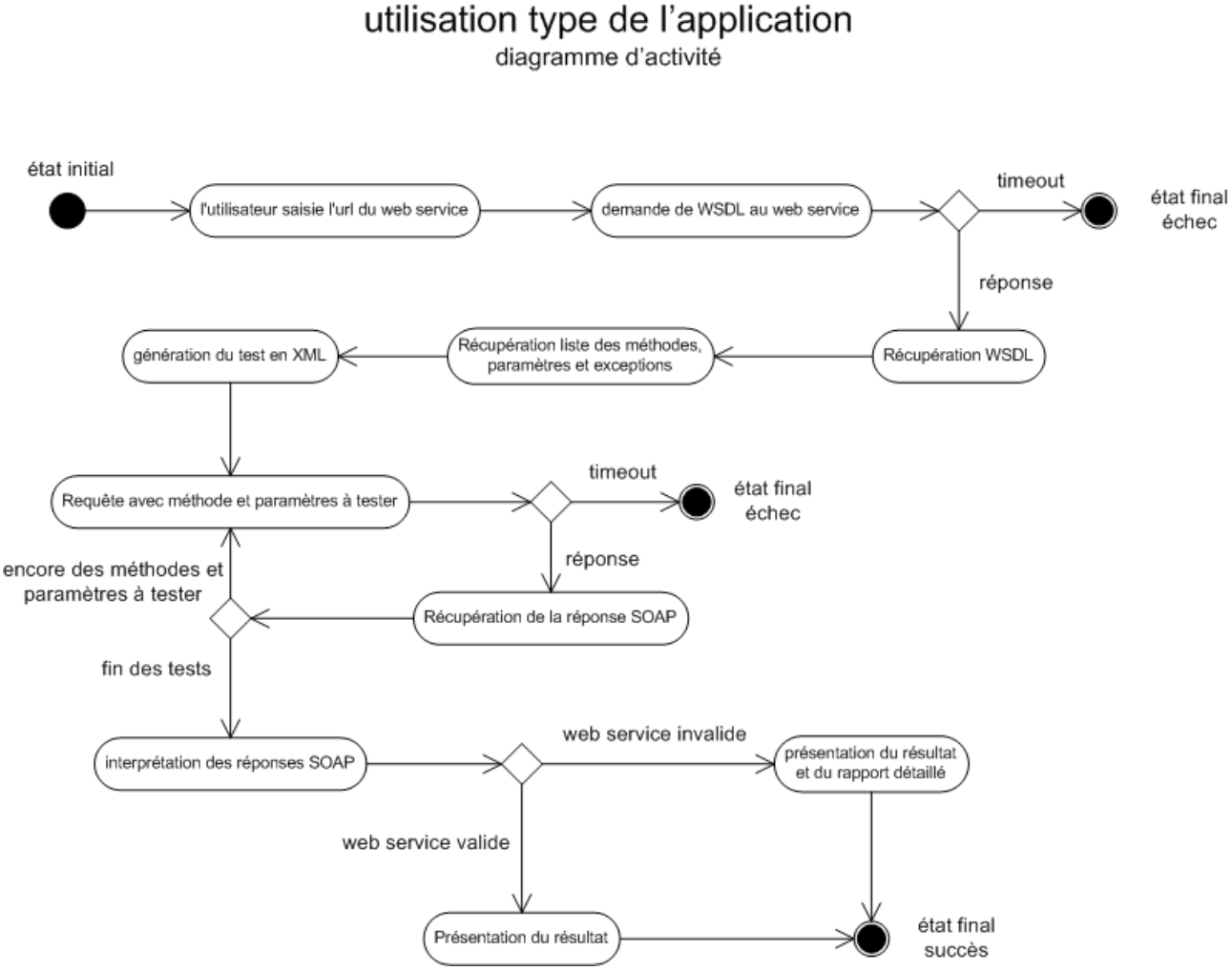
6) L'application propose à l'utilisateur la saisie des paramètres de la méthode à tester.

7) L'utilisateur saisie les paramètres et lance la validation.

8) L'application teste la méthode à tester, avec les paramètres saisis par l'utilisateur, par une requête au web service.

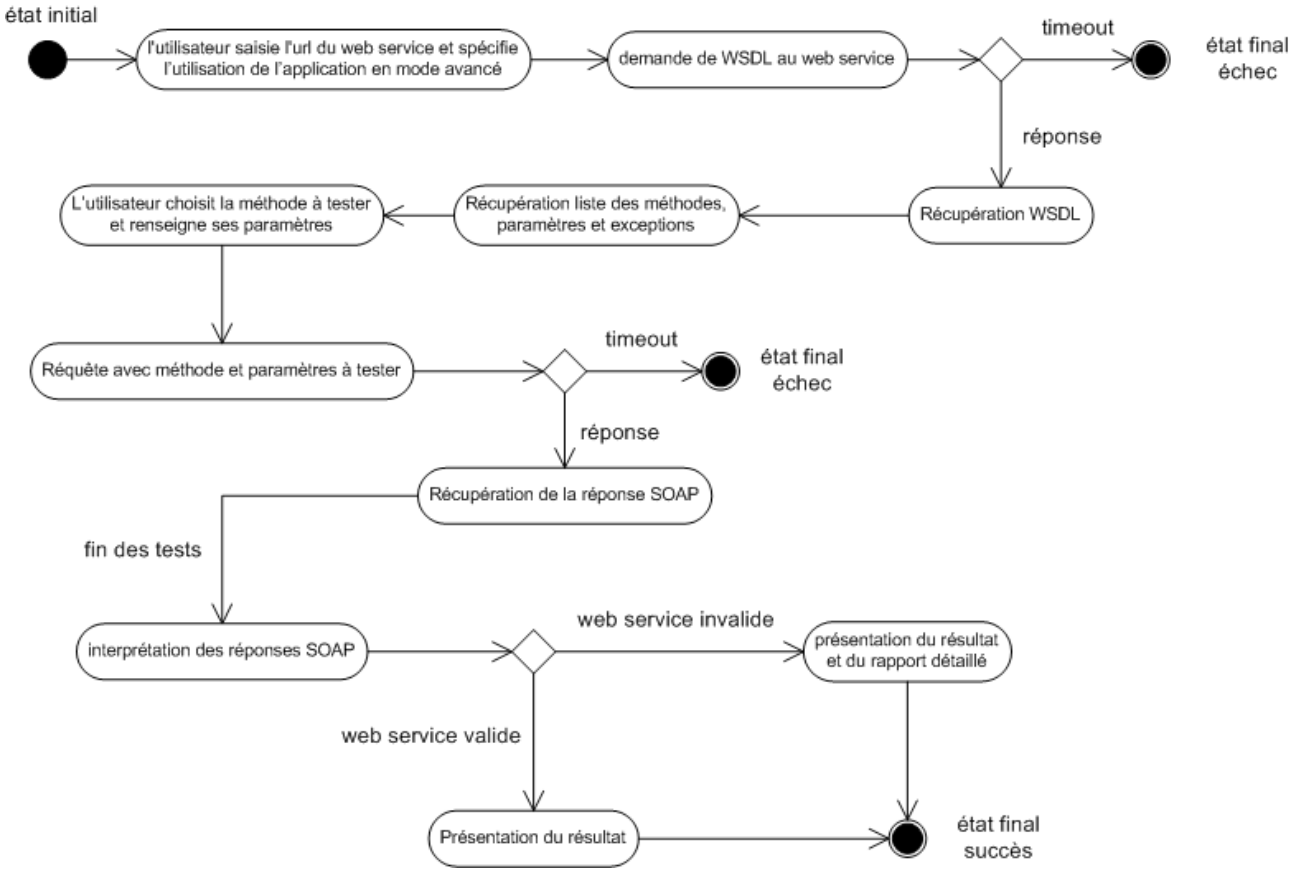
9) Le web service répond en SOAP à la requête.

10) L'application communique à l'utilisateur la validité de la méthode du web service avec les paramètres qu'il a saisis, et lui présente un rapport en cas de non validation.



- *L'utilisateur renseigne l'application sur l'url du Web-Service.*
- *L'utilisateur lance la validation.*
- *L'application récupère la description WSDL du Web-Service.*
- *L'application récupère la liste des méthodes et paramètres à partir du WSDL.*
- *L'application génère un jeu d'essais en XML pour le Web-Service.*
- *L'application test son jeu d'essais par des requêtes au Web-Service.*
- *L'application reçoit et interprète les résultats.*
- *L'application renseigne l'utilisateur sur la validité du Web-Service, et lui présente un rapport en cas de non-validation.*

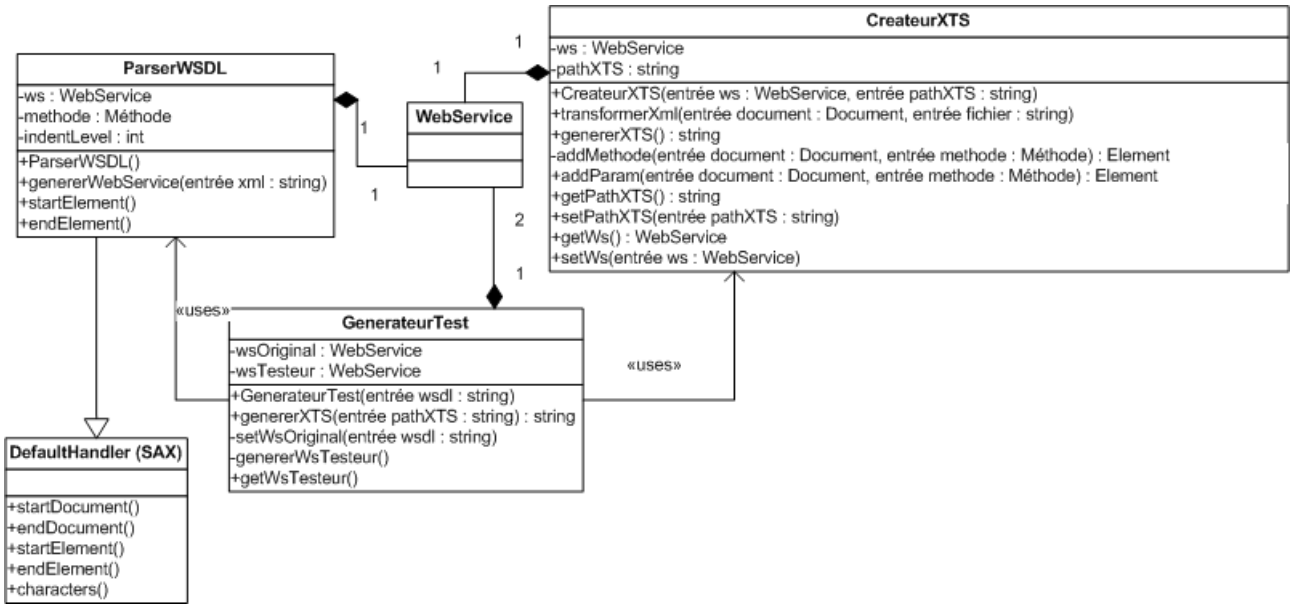
utilisation avancée de l'application
diagramme d'activité



- *L'utilisateur renseigne l'application sur l'url du Web-Service.*
- *L'utilisateur spécifie qu'il souhaite utiliser l'application de manière avancée.*
- *L'utilisateur lance la récupération de la description WSDL.*
- *L'application récupère la description WSDL du Web-Service.*
- *L'application extrait les méthodes et les paramètres de la description WSDL et les présente à l'utilisateur.*
- *L'utilisateur choisit la méthode à tester du Web-Service.*
- *L'utilisateur renseigne les paramètres à utiliser.*
- *L'utilisateur lance la validation de la méthode choisie.*
- *L'application test le jeu d'essais de l'utilisateur par une ou des requêtes au Web-Service.*
- *L'application reçoit les résultats.*
- *L'application interprète les résultats.*
- *L'application renseigne l'utilisateur sur la validité de la méthode du Web-Service avec les paramètres que l'utilisateur a saisis, et lui présente un rapport en cas de non-validation.*

Conception

ParserWSDL et CreateurXTS :

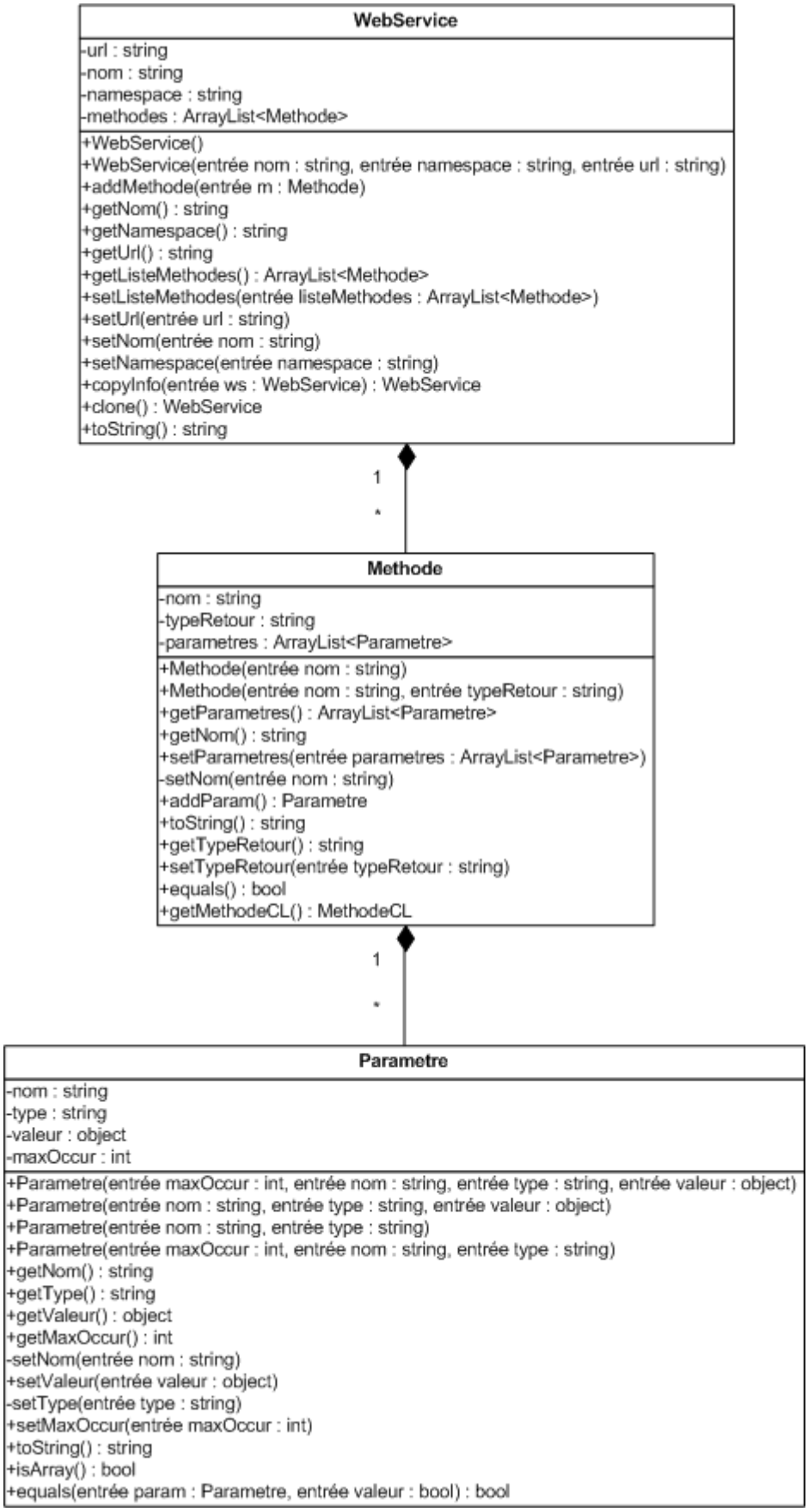


Voici le détail des classes ParserWSDL et CreateurXTS, toutes deux utilisées par le GenerateurTest pour créer un schéma XML de test à partir de la description WSDL du web service distant.

ParserWSDL permet, grâce à son héritage de la classe DefaultHandler de SAX, de parser un fichier XML. Ici ParserWSDL est spécialisé dans le parsing de description WSDL, ensuite convertit en objet WebService, ce dernier étant également détaillé plus loin. C'est GenerateurTest qui utilise ParserWSDL dans le but de récupérer un WebService pour lequel il pourra générer des tests.

Le CreateurXTS, quant à lui, est aussi utilisé par le GenerateurTest : ce dernier lui fournit les tests à effectuer sous forme d'objet WebService. Le rôle de CreateurXTS, qui utilise des classes de DOM, est de transformer l'objet WebService du GenerateurTest, en fichier de tests XML, selon le schéma XML de test XTS traité plus loin dans ce rapport. Il va générer une description du Web Service distant, de ces méthodes et des paramètres de celle-ci, ainsi qu'un jeu d'essais pour chaque méthode.

L'objet WebService :



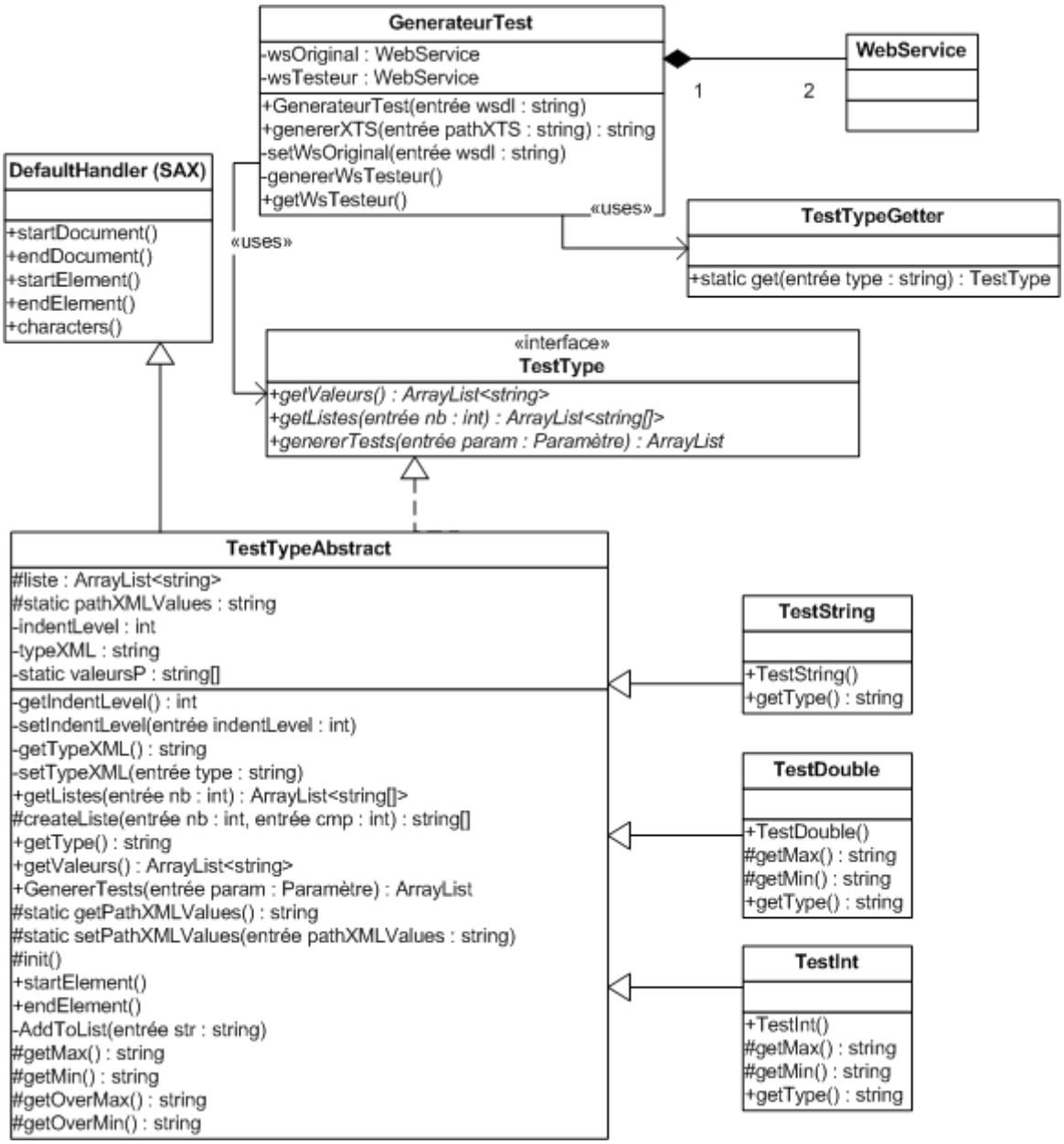
Un objet WebService est une représentation d'un Web Service distant défini, grâce notamment à son url, son nom et son namespace. Il dispose également d'une collection des méthodes du Web Service distant, représentée par une collection d'objet Methode.

Chaque Methode possède les particularités de la méthode du Web Service distant, comme son nom et son typeRetour. Les Methodes possèdent également une collection de Parametres, objets qui représentent les paramètres des méthodes du Web Service distant.

Les paramètres peuvent contenir des valeurs (Object) qui permettent à l'objet WebService de mémoriser des tests à effectuer sur le Web Service distant. L'attribut maxOccur permet de spécifier, dans le cas d'un tableau, le nombre maximal d'éléments du tableau supporté par la méthode du Web Service distant.

On peut remarquer la présence de la méthode Methode getMethodeCL() dans la classe Methode, qui sert à copier la Methode courante dans un objet de type MethodeCL. L'objet MethodeCL est la copie exacte de l'objet Methode, à la différence que ses Parametres ne sont pas stockés dans une ArrayList, mais dans un tableau de String (au lieu d'un tableau d'objet), dans le but d'un envoi de la Methode en SOAP à un client dans l'optique d'une utilisation avancée de l'application.

Le GenerateurTest :



Le GenerateurTest est une des parties clés de l'application. En effet c'est lui qui a la responsabilité de générer des tests pour les méthodes du Web Service distant. Le GenerateurTest ne fonctionne pas tout seul, il utilise plusieurs autres classes. Voici comment son mécanisme pourrait être résumé :

- Le GenerateurTest va récupérer une représentation du Web Service distant à partir de la description WSDL de ce dernier.
- Pour chaque méthode du Web Service distant à tester, le GenerateurTest découpe la génération des tests en sous-tâches.
- Ces sous-tâches sont ensuite réparties à des classes spécialisées compétentes.
- Le GenerateurTest va ensuite assembler les réponses des classes spécialisées et en créer une représentation en mémoire, soit un objet WebService.
- Pour finir le CreateurXTS va sauvegarder ces tests dans un fichier XML.

Pour comprendre plus en détails ce mécanisme, intéressons nous aux classes spécialisées et à leur fonctionnement :

Une classe spécialisée, compétente pour traiter une sous-tâche de génération de tests, est une classe qui hérite de la classe abstraite TestTypeAbstract. Celle-ci hérite la classe DefaultHandler de SAX, et implémente l'interface TestType. Chacune de ces entités se situent dans le package testType de l'application.

Chaque classe est spécialisée et compétente pour traiter des sous-tâches de génération de tests pour un type de paramètre. Il y a trois classes pour trois types de paramètres : TestInt, TestDouble et TestString, respectivement spécialisée pour les traitement de sous-tâches concernant les paramètres de types int, double et String.

Si on remonte dans la hiérarchie des classes on retrouve la classe abstraite mère des précédentes : TestTypeAbstract.

Cette classe, grâce à son héritage de la classe DefaultHandler de SAX, peut effectuer le parsing d'un fichier essentiel à nos classes spécialisés : Il s'agit du fichier xml ValeursTest.xml, contenu dans le package testType. Le rôle de ce fichier est très important car c'est lui qui va spécifier les types de tests à effectuer pour les différents types de paramètres. Son intérêt est évident, car il permet une modification et une remise en cause des valeurs particulières de test pour chaque type de paramètre, le tout sans pré-compiler l'application pour prendre en compte les changements.

La configuration du fichier valeursTest.xml se veut simple, et propose en plus certaines valeurs particulières non statiques et/ou dépendantes des types de paramètres pour aller plus loin, mais aussi pour simplifier la configuration. Pour tenter de mieux comprendre son principe, en voici un extrait avec la définition des valeurs spéciales ainsi que les valeurs pour le type de paramètre double :

```
<!--
Vous pouvez complétez les valeurs qui seront testé par le GenerateurTest en fonction des
types de variables que la méthode du WebService attend.
Un certain nombre de valeurs spéciales on été prévues :
Valeurs spéciales:
Elles sont à écrire entre parenthèses
En voici la liste :
                (intRand+) :            entier aléatoire positif
                (intRand-) :            entier aléatoire négatif
                (doubleRand+) :         double aléatoire positif
                (doubleRand-) :         double aléatoire négatif
                (string8192) :          chaine de 8192 caractères divers
                (null) :                valeur null
                (min) :                 valeur minimun pour le type
                (max) :                 valeur maximun pour le type
                (overMin) :             en dessous de la valeur minimun pour le type
                (overMax) :             au dessus de la valeur maximun pour le type
                (infinity+) :           valeur particulière Double.POSITIVE_INFINITY
                (infinity-) :           valeur particulière Double.NEGATIVE_INFINITY
 -->
<valeurs-tests>
                ...
                <type id="double">
                        <!-- valeurs testé pour les types double -->
                        <valeur value="0"/>
                        <valeur value=""/>
                        <valeur value="hello world"/>
                        <valeur value="(doubleRand+)"/>
                        <valeur value="(doubleRand-)"/>
                        <valeur value="(null)"/>
                        <valeur value="(min)"/>
                        <valeur value="(max)"/>
                        <valeur value="(overMin)"/>
                        <valeur value="(overMax)"/>
                        <valeur value="(infinity+)"/>
                        <valeur value="(infinity-)"/>
                </type>
                ...
</valeurs-tests>
```

Pour le double on teste ici avec un zéro, une chaîne vide, une chaîne de caractères, un double aléatoire positif, un double aléatoire négatif, un null, la valeur minimum pour le type, donc la valeur minimum pour un double, la valeur maximum pour un double, un débordement avec une valeur supérieure au max du type, donc du double, puis un débordement avec une valeur inférieure au minimum du double, et pour finir les valeurs spéciales qui représentent l'infini positif et l'infini négatif pour un double.

La plupart des opérations sont effectuées par la classe abstraite TestTypeAbstract, et ses classes filles permettent une spécialisation par type de paramètres.

L'interface TestType permet de définir les méthodes que doivent implémenter les classes de traitement de sous-tâches de génération de test. Ces méthodes sont au nombre de trois et permettent la génération :

- d'une collection de valeurs simples à tester
- d'une collection de valeurs de types complexes (tableaux) à tester
- d'une collection de valeurs à tester, cette collection étant une collection de valeurs simples ou complexes à tester, en fonction de l'objet de type Parametre pour lequel il faut générer une collection de valeurs de tests.

Le GenerateurTest délègue donc la génération de tests à des objet implémentant TestType. Mais le GenerateurTest ne doit pas avoir à s'occuper de la gestion de ces objets, c'est pourquoi il utilise la classe TestTypeGetter. Cette dernière fourni au GenerateurTest une instance de la classe spécialisé compétente pour le type de paramètre que le GenerateurTest lui aura demandé.

Une fois les sous-tâches effectuées, le GenerateurTest rassemble les différents tests de chaque paramètre par méthode du Web Service afin de pouvoir effectuer plusieurs tests par méthode.

Au début nous avons envisagé un produit cartésien des valeurs de chaque paramètre, afin de pouvoir tester tous les cas possibles, mais un rapide calcul comme le suivant nous a fait changer d'avis :

Supposons quatorze valeurs à tester pour un paramètre de type int et douze pour un paramètre de type double. Ces nombres de valeurs sont dépendants du fichier xml de configuration des valeurs test et sont les nombres de valeurs actuellement en place dans ce fichier. Imaginons un Web Service à tester qui dispose d'une méthode prenant comme paramètres un tableau de int, de dix valeurs maximum, et un double, par exemple String maMethode(int[], double). En produit cartésien on obtiendrait :
*([nb valeur int] ^ [taille max tableau]) * [nb valeurs double] tests*
*ce qui donnerais pour ce cas de figure (14^10)*12 = 3 471 055 859 712 tests.*
Si l'on rajoute le fait que pour les tableaux, les classes spécialisés de TestTypeAbstract génèrent un jeux d'essai avec la taille [taille max du tableau], et un deuxième jeux d'essais, identique au premier, avec un tableau de taille [taille max du tableau + 1], le calcul deviendrait alors :
*((14^10) * 12) + ((14^11) * 12) = 3 471 055 859 712 + 48 594 782 035 968 tests*
soit 52 065 837 895 680 tests (plus de 52 065 milliards de tests).

Supposons qu'un test se fasse en moyenne en 0,1 seconde, le temps d'envoyer la requête au Web Service, que celui-ci traite la requête et renvoie son résultat, et que notre application récupère ce résultat. Cette supposition est très optimiste d'après nos tests. Il faudrait alors 52 065 837 895 680 / 3 600 / 24 / 365 ans
Soit 1 650 996 ans pour tester seulement cette méthode ne comportant en paramètres qu'un simple tableau de 10 int et un double.

Sans considérer les théoriques autres méthodes du Web Service à tester, ce calcul nous montre que la tâche est inenvisageable.

La démarche adoptée a donc été la suivante :
Prenons une méthode sans type complexe pour simplifier, donc aucun tableau:
String maMethode(int, double, double).
Le GenerateurTest va récupérer le nombre de tests pour chaque paramètre de la méthode, sois 14 pour le int, 12 pour le premier double, et 12 pour le second double.
Il va conserver le nombre maximum, ici 14. Il va donc effectuer, pour notre cas, quatorze tests pour la méthode grâce à une boucle de cette forme :

```
pour i allant de 0 à 14
        valeurInt = testInt.valeur[ i modulo testInt.nbValeur ];
        valeurDouble1 = testDouble.valeur[ i modulo testDouble.nbValeur ];
        valeurDouble2 = testDouble.valeur[ i modulo testDouble.nbValeur ];
        enregistrerLeTest();
fin pour
```

En fait, afin d'augmenter la pertinence du test, la boucle suivante va pouvoir, dans la plupart des cas, être effectuée deux fois. Voici l'algorithme qui définit le nombre de boucles:

```
maxValeur = nombre de valeurs de test maximum des paramètres à tester. (ici 14)
nb_boucle = 1;
Pour chaque paramètre
        si maxValeur modulo paramètre.nbValeur != 0
        // Si nbValeur du paramètre n'est pas un multiple de maxValeur
        alors nb_boucle = 2;
fin pour.
//puis la boucle déjà vu
pour i allant de 0 à (14 * nb_boucle)
        valeurInt = testInt.valeur[ i modulo testInt.nbValeur ];
        valeurDouble1 = testDouble.valeur[ i modulo testDouble.nbValeur ];
        valeurDouble2 = testDouble.valeur[ i modulo testDouble.nbValeur ];
        enregistrerLeTest();
fin pour
```

Cet algorithme permet de doubler le nombre de tests à effectuer à condition que les valeurs de la seconde boucle ne seront pas celles de la premières. Pour illustrer cela, voici un exemple avec dans le premier cas deux types, dont premier.nbValeur = 2 et second.nbValeur = 4 (donc maxValeur = 4).

Si l'on effectue deux fois la boucle, voici un exemple de test pouvant être généré :

a,1
b,2
a,3
b,4 *//fin première boucle*
a,1
b,2
a,3
b,4 *//fin seconde boucle*

La deuxième boucle donne ici les mêmes tests que pour la première.

Prenons maintenant premier.nbValeur = 3 et second.nbValeur = 4 (donc maxValeur = 4).

a,1
b,2
c,3
a,4 *//fin première boucle*
b,1
c,2
a,3
b,4 *//fin seconde boucle*

On constate ici que les tests de la seconde boucle ne sont pas les mêmes que pour la première. Le choix d'une deuxième boucle est donc dans ce cas intéressant pour augmenter le nombre de tests de façon limitée, mais pertinente.

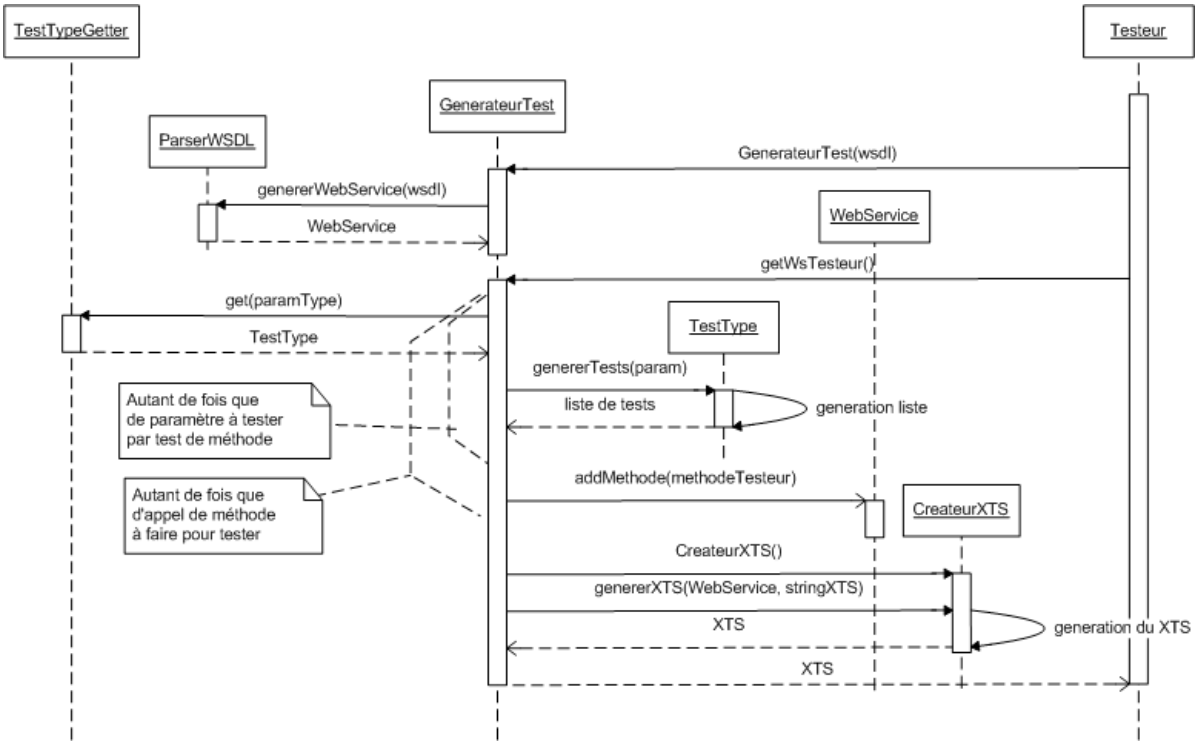
La logique pour les valeurs des tableaux est très simple: un tableau sera rempli une fois avec chaque valeur test de son type, dans un tableau de la taille maximum spécifiée. Puis une deuxième fois avec chaque valeur test de son type, mais dans un tableau de la taille maximum spécifiée + 1.

Si l'on prend pour un tableau d'entier la liste de valeurs : 1, 2, 3
et que la taille maximum du tableau est de 4, voici les valeurs de tests du tableau :
[1,1,1,1]
[2,2,2,2]
[3,3,3,3]
[1,1,1,1,1]
[2,2,2,2,2]
[3,3,3,3,3]

Pour chaque test de méthode générée, le GenerateurTest va stocker le test de la méthode sous forme d'objet Methode dans un objet WebService. Une fois tous les tests ajoutés à l'objet WebService pour toutes les méthodes du WebService distant, le GenerateurTest va demander au CreateurXTS la génération d'un schéma XML de test: un fichier XTS.

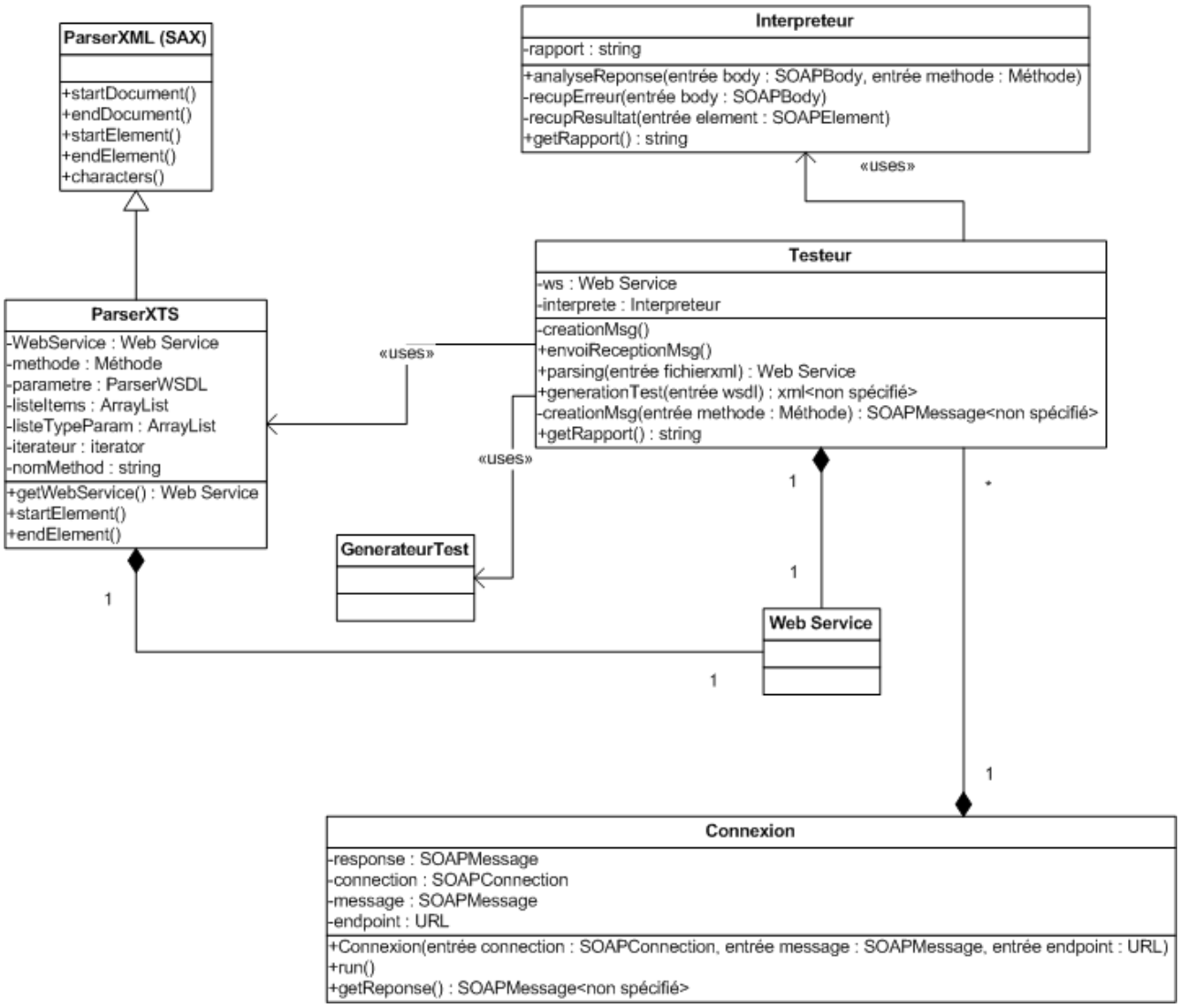
On peut s'interroger sur la nécessité de passer par un format de fichier XML intermédiaire : L'application va devoir, dans un premier temps, le créer à partir d'un objet WebService. Puis l'application va devoir parser le fichier XML, afin de recharger le WebService en mémoire.
Cette conversion présente en fait deux avantages pertinents :
Le premier est de pouvoir écrire des tests à la main en créant, ou modifiant, un schéma XML de tests.
Le deuxième avantage est l'obtention d'une bonne modularité de l'application qui peut donc fonctionner en deux parties indépendantes l'une de l'autre, ceci pouvant être très pratique en vue d'un théorique redéveloppement futur d'une partie de l'application, ou de l'ajout de modules...

Diagramme de Séquence sur la génération de tests :



Le Testeur crée une instance de GenerateurTest, en lui fournissant le chemin de la description WSDL du Web Service distant à tester. Le GenerateurTest fait passer le chemin de la description WSDL à une instance de ParserWSDL qu'il crée pour l'occasion. Le ParserWSDL renvoie au GenerateurTest une instance de WebService qui représente la structure du Web Service distant à tester. Le Testeur demande au GenerateurTest un schéma XML de tests, le GenerateurTest doit donc générer ces tests. Pour cela le GenerateurTest utilise la méthode static de TestTypeGetter qui lui fournit une classe TestType spécialisée pour le type de paramètres à traiter, et cela pour chaque paramètres de chaque tests de méthode. Chaque test est ajouté à un objet WebService sous forme de Methode aux valeurs de Parametres renseignées. Le GenerateurTest va effectuer ces opérations pour chaque test de chaque méthode du Web Service distant. Une fois que le WebService à tester est renseigné, le GenerateurTest va créer un CreateurXTS et lui fournir le WebService à transformer en schéma XML de tests. Le GenerateurTest récupère ensuite le schéma tout juste créé et l'envoi au Testeur.

Le Testeur :



Le but du Testeur est d 'effectuer les tests fournis par le générateur de Test à travers un fichier XTS.

Pour cela il utilise plusieurs autres classes :

ParserXTS :

Cette classe lui permet de générer un objet WebService à partir d'un fichier XTS.

Cet objet WebService contient une liste d'objets Methode qui symbolise la liste des tests à effectuer.

La classe hérite de la classe DefautHandler qui permet de parser un document XML.

En effet chaque fois que le parser rencontre une balise ouvrante ou fermante il déclenche respectivement les méthodes startElement et endElement. Ces méthodes permettent de connaître le nom, la valeur, les attributs du noeud. Ainsi c'est à grâce à ces éléments que la classe construit un WebService.

Connexion :

Cette classe permet d'envoyer un message Soap, via une objet SOAPConnection à l'url désirée qui est l'url du Web Service et d'attendre la réponse.

Afin justement que toute l'application ne soit bloquée par cette attente, cette classe est un thread. Ainsi le Testeur construit cette classe avec les paramètres cités plus haut : SOAPConnection, l'url du web service, SOAPMessage. Ensuite il démarre ce thread via start(). C'est à ce moment que la classe Connexion lance la connexion, le thread est alors bloqué jusqu'à l'arrivée de la réponse. Une fois la connexion lancée le Testeur va effectuer périodiquement la méthode getResponse() de Connexion afin de savoir si la réponse est arrivée et le cas échéant la récupérer afin de continuer le traitement. Si au bout d'un certains temps la réponse n'est toujours pas obtenue, le Testeur continuera quand même son traitement.

Interpréteur :

Cette classe analyse les réponses que lui fournit le Testeur via la méthode AnalyseReponse.

En réalité, le Testeur ne lui fournit que le corps de la réponse (SOAPBody) ainsi que la méthode testée afin de connaître les paramètres entrés pour les insérer dans le rapport.

L'interpréteur teste tout d'abord si le SOAPBody est nul. Si c'est le cas cela signifie que le Service Web n'a pas répondu assez rapidement à la requête (le testeur n'a pas attendu la réponse et à transmis null à l'interpréteur) et l'interpréteur insère un message en conséquence dans le rapport.

Si le SOAPBody n'est pas nul et donc qu'une réponse a été obtenue, l'interpréteur vérifie s'il y a une erreur en utilisant la méthode hasFault de SOAPBody qui rend true s'il y a une erreur. Dans ce cas l'interpréteur transmet le SOAPBody à la méthode recupErreur qui a pour fonction d'interpréter les erreurs SOAP. Elle insère son interprétation dans le rapport.

Si le SOAPBody n'est pas nul et ne contient pas d'erreur, cela signifie qu'un résultat

a été obtenu. Dans ce cas l'interpréteur fait appel à sa méthode recupResultat qui interprète le résultat et l'insère dans le rapport.

Les tests sont créés et envoyés par le Testeur via les méthodes envoiReceptionMsg et creationMsg.

La première parcourt les méthodes de l'objet WebService, initialise une connexion, envoie le message (via l'objet Connexion) créé par creationMessage, récupère la réponse et la transmet à l'interpréteur.

CreationMsg crée le message qui sera envoyé. Elle utilise pour cela un objet Methode qui correspond au test à effectuer.
Via la classe MessageFactory (provenant de l'API SAAJ) elle crée un message vide mais qui contient la structure d'un message SOAP : enveloppe, header, body :

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
		<env:Header/>
		<env:Body></env:Body>
</env:Envelope>
```

Ensuite la méthode ajoute un enfant au body qui correspond à la méthode à tester
Ce qui donne dans le code :

```
QName bodyName = new QName(methode.getNom());
SOAPBodyElement bodyElement =body.addBodyElement(bodyName);
```

Et pour le message SOAP :

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
		<env:Header/>
		<env:Body>
			<carre></carre>
		</env:Body>
</env:Envelope>
```

Puis, il faut lui ajouter les paramètres de la méthode. Pour cela on ajoute au SOAPElement qui correspond à la methode (bodyElement) un enfant pour chaque paramètre.

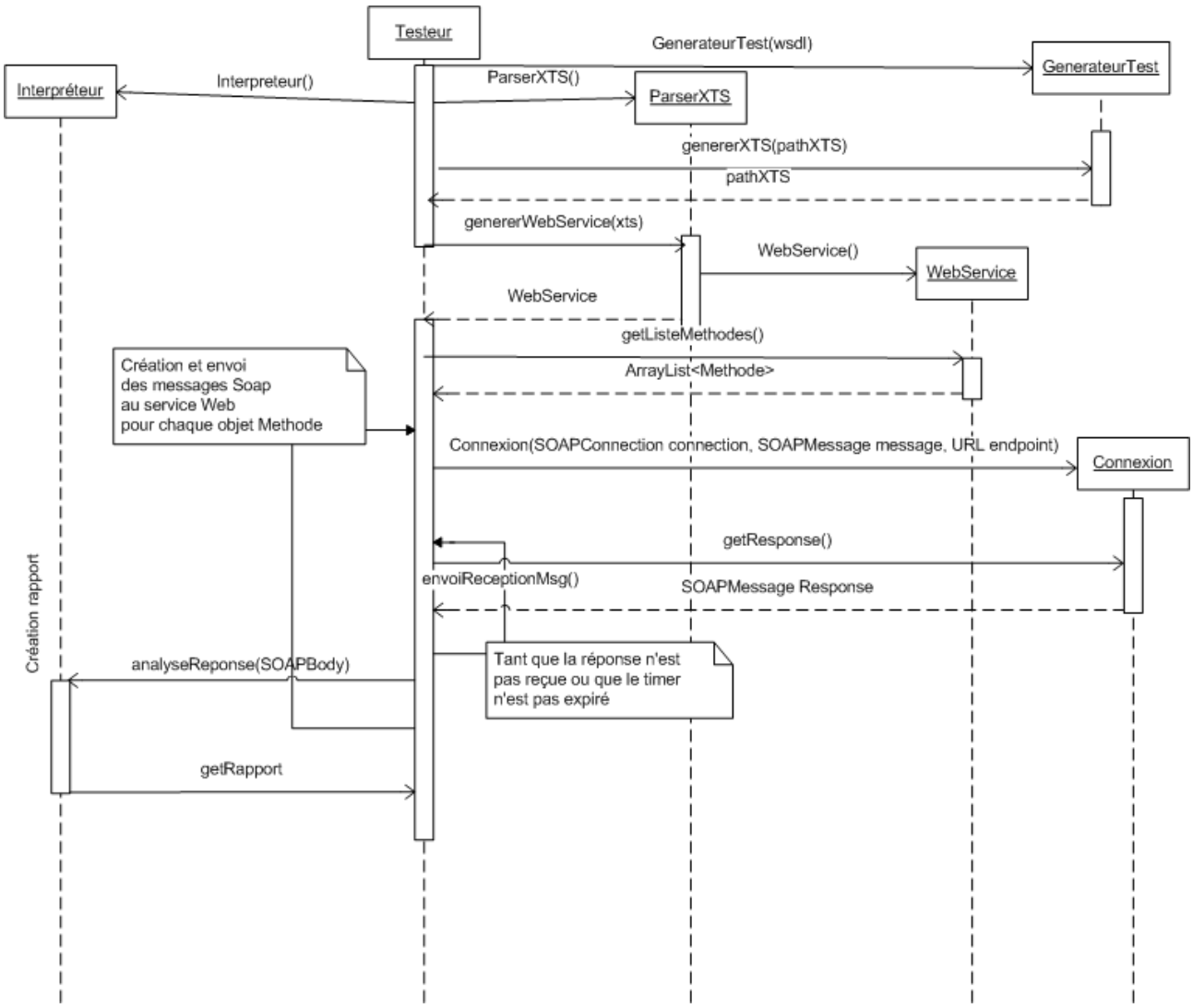
Ce qui donne dans le code (pour chaque paramètre):

```
//nom est le nom du paramètre
//valeur est la valeur du paramètre
QName paramName = new QName(nom);

SOAPElement elem =
bodyElement.addChildElement(paramName);
 elem.addTextNode(valeur.toString());
```

Et pour le message SOAP :

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
        <env:Header/>
        <env:Body>
                <carre >
                        <x>2147483647</x>
                </carre>
        </env:Body>
</env:Envelope>
```



Le Testeur utilise la classe ParserXTS qui à partir d'un fichier XTS crée un objet WebService.

Puis via la méthode envoiReceptionMsg() il parcourt la liste des méthodes et pour chaque méthode crée une requête SOAP (via la classe creationMsg()). Cette requête est ensuite envoyée via un thread (classe Connexion) afin de ne pas bloquer totalement l'application si le serveur ne répond pas. Ensuite envoiReceptionMsg récupère la réponse et la transmet à la classe Interpréteur.

Cette classe analyse la réponse, établit s'il y a une erreur ou non et ajoute le résultat de son analyse au rapport de test.

Voici à quoi peut ressembler un rapport :

```
Résultat du test sur la méthode inverseTab:
En entrée :
tab (int[]) : <tableau>
0
0
0
0
0
0
0
0
0
0
</tableau>
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
inverseTabReturn : 0
--------------------------------
Résultat du test sur la méthode inverseTab:
En entrée :
tab (int[]) :
<tableau>

</tableau>
Une erreur est survenue :
Contenu de l'erreur Soap:
  code erreur = soapenv:Server.userException
  Local name = Server.userException
  Namespace prefix = soapenv, bound to http://www.w3.org/2003/05/soap-
envelope
  Fault string = java.lang.IllegalArgumentException
  Detail entry = badconker
--------------------------------
Résultat du test sur la méthode inverseTab:
En entrée :
tab (int[]) : <tableau>
```

```
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
</tableau>
Une erreur est survenue :
Contenu de l'erreur Soap:
  code erreur = soapenv:Sender
  Local name = Sender
  Namespace prefix = soapenv, bound to http://www.w3.org/2003/05/soap-
envelope
  Fault string = string
  Detail entry = badconker
--------------------------------
Résultat du test sur la méthode inverseTab:
En entrée :
tab (int[]) : <tableau>
2026336827
2026336827
2026336827
2026336827
2026336827
2026336827
2026336827
2026336827
2026336827
2026336827
</tableau>
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
inverseTabReturn : 2026336827
--------------------------------
```

Structure d'un fichier XTS :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<webservice name="http://services" namespace="MonWebService"
url="http://localhost:8080/testWebService/services/MonWebService">
		<method name="carre" return="int">
			<typesAttributes>
				<typeAttribute type="int" name="x" />
			</typesAttributes>
			<tests>
				<attributes>
					<attribute value="32.4"/>
				</attributes>
				<attributes>
					<attribute value="0"/>
				</attributes>
				<attributes>
					<attribute value="65535"/>
				</attributes>
				<attributes>
					<attribute value="-65534"/>
				</attributes>
			</tests>
		</method>
		<method name="tableauString" return="string">
			<typesAttributes>
				<typeAttribute type="string[]" name="s" />
			</typesAttributes>
			<tests>
				<attributes>
					<attributeComplex>
						<item value="un"/>
						<item value="deux"/>
						<item value="trois"/>
						<item value="quatre"/>
					</attributeComplex>
				</attributes>
			</tests>
		</method>
</webservice>
```

Un web service est nommé, il est situé à une url et possède un namespace, attributs qu'il possède dans ses balises <webservice></webservice>.
Le web service possède aussi des méthodes à tester.
Une méthode, représentée par les balises <methode></methode> possède un nom et un

type de retour définis en attributs. La méthode possède également des paramètres définis grâce à ses balises <typeAttributes></typeAttributes>.
Une méthode possède plusieurs tests: pour éviter la redondance, le contenu des balises <attributes></attributes> représente les paramètres d'un des tests de la méthode, et tous les tests sont regroupés par méthodes.
Les attributs des balises <attribute></attribute> représentent le type, le nom et la valeur d'un des paramètres d'un des tests de la méthode du Service Web.
On constate aussi les balises

```
<attributeComplex>
                    <item value="un"/>
                    <item value="deux"/>
                    <item value="trois"/>
                    <item value="quatre"/>
</attributeComplex>
```

qui permettent de définir des type complexes (tableaux) et leurs contenus.

Conclusion

Au terme de notre développement, l'objectif principal a pu être réalisé, à savoir la réalisation d'une application fonctionnelle de validation de Web Services.

En effet notre application, un Web Service en l'occurrence, remplit son rôle, c'est à dire qu'elle vérifie les Web Service par rapport à leur description, et elle leur impose une batterie de tests afin de vérifier leur robustesse et leur gestion des erreurs.

Ce projet, de par son architecture modulaire, nous a permis de réfléchir à une méthode de travail. Celle-ci nous permettant de gérer une architecture importante, évolutive respectant les « best-practices », tout en nous coordonnant efficacement.

Après une introduction aux technologies des Web Services en cours, ce projet a été un bon prolongement. En effet il nous a permis d'approfondir ces nouvelles technologies dans des domaines variés : SOAP avec SAAJ, le parsing XML avec SAX et DOM, et le fonctionnement des Web Services en fonction de leurs différents protocoles de communication SOAP.

Pour conclure, la modularité de notre application permettra son amélioration et l'ajout de modules très aisément.

Bibliographie

Voiçi une liste de nos principales sources pour les différentes phases du projet et pour la rédaction du rapport :

Magazine Programmez! : www.programmez.com

Wikipedia, l'encyclopedie libre : www.wikipedia.org

Site Web officiel du Java : http://java.sun.com

Site Web de Tomcat : http://tomcat.apache.org

Tutoriel Web Service de Sun : http://java.sun.com/webservices/docs/2.0/tutorial/doc/index.html

Et bien sûr les cours de Web Service de M. Salva pour les licences pro web.

Annexes

Voici une capture d'écran du client en swing (composant graphique java) qui permet la consommation de notre Web Service. Il comporte un champ url qui permet de renseigner l'adresse de la description wsdl du Web Service à tester. On choisit ensuite la validation automatique, ou manuelle, avec les boutons juste en dessous. Dans le cas d'une validation manuelle, le client nous propose une liste des méthodes disponibles. Après sélection de cette dernière le client nous propose un champ de saisie par paramètre à renseigner. Une fois ces champs remplis, l'utilisateur peut lancer le test de sa méthode avec le bouton envoyer. Le rapport du test s'affichera dans la zone texte en bas du client. Dans le cas d'une validation automatique, un clique sur le bouton validation automatique lance la validation et affiche le rapport en dessous une fois les tests effectués.