M2 CCI – EP1 Algorithmique

TP5 (4h) – 25 oct. 2017

But de la séance : Traitements de chaînes de caractères

Le contexte de cet exercice est l'administration de site internet, coté serveur, en particulier le suivi des connexions à travers la journalisation effectuée par le serveur. En effet, le serveur web remplit plusieurs fonctions annexes au service des pages. L'une d'entre elles concerne la journalisation. Lorsque le serveur reçoit une demande, il inscrit dans un fichier cette demande et comment il y a répondu. L'analyse du contenu de ce fichier permet de savoir ce qui est demandé, comment, quand, depuis où, par qui. En rassemblant toutes ces informations sur une plage de temps donnée, on peut reconstruire la session d'une machine, c'est à dire retracer pour une adresse IP source donnée dans quel ordre et à quel rythme les demandes sont faites. Cela permet d'établir un profil d'utilisation du site, c'est à dire essayer de catégoriser les motivations du demandeur. On pourra aussi constater les tentatives d'attaque, ainsi que les dysfonctionnements éventuels. Mais avant d'en arriver là, y a du travail !

Vous disposez comme données d'entrée d'un extrait de fichier log (à télécharger sur le site du cours en ligne). Ce fichier a été produit par le serveur Apache. Vous trouverez une documentation complète sur le contenu de ce fichier sur le site http://httpd.apache.org/docs/trunk/fr/logs.html. Chaque ligne correspond au traitement d'une requête reçue par le serveur. Le format typique d'une de ces lignes est donné en annexe. Cette annexe suffit pour l'exercice.

Pour ce premier volet, vous définirez dans votre programme plusieurs chaînes de caractères avec comme contenu l'une des lignes du fichier log (en faisant un copier-coller). Par exemple :

```
String s = "195.242.73.130 - - [23/Oct/2003:10:26:48 +0200] \"GET / HTTP/1.1\" 200 324 \"-\" \"Mozilla/4.0 (compatible; MSIE 5.5; Windows 98; Win 9x 4.90)\"\n";
```

Vous noterez la présence des anti-slashes (\) permettant d'inclure dans la chaîne des guillemets sans interférer avec la syntaxe de Java. Ces chaînes permettront de tester le programme dans différents cas de figure. Les questions du TP s'appuient sur la valeur de la chaîne ci-dessus. C'est également elle qui est prise en exemple en annexe.

Le but de cette première partie est de découper la chaîne de façon à en extraire les champs tels que définis en annexe. La tâche est un peu plus complexe qu'il n'y paraît car certains champs ont une longueur fixe et et sont délimités par des caractères bien définis tandis que pour d'autres on ne sait *a priori* le nombre de caractères qui vont les composer.

Le TP est formé de plusieurs parties. La première est composée de quatre questions et correspond aux traitements de base qu'il faut savoir réaliser pour analyser l'ensemble de la chaîne, même si l'ensemble n'est pas traité dans ces quatre questions.

La seconde partie permet de formaliser la requête sous la forme d'un objet qui servira, par la suite, à faire divers traitements, en particulier des statistiques. Cela permet d'étendre le sujet dans d'autres direction, en particulier la

structuration du code. Enfin la dernière partie, pour les plus avancés, sort du cadre et montre une autre approche, plus systématique, du décodage des chaînes au moyen d'automates à état. On se rapproche ainsi de la théorie des algorithmes et notamment de leur formalisation au moyen des machines de Turing.

Les exercices seront traités dans l'ordre, l'important n'est pas le nombre réalisé à la fin de la séance, mais votre compréhension et votre capacité à le refaire.

Table des matières

Traitements de base

Question 1

Ecrire une méthode **getAdresseIP** qui renvoie, sous la forme d'une chaîne, le premier champ (adresse IP) en commençant par calculer la position du premier espace rencontré dans la chaîne, puis en recopiant la partie de la chaîne située entre le début et cet espace.

Question 2

Ecrire une méthode **get_DateRequete** qui renvoie sous la forme d'une chaîne le champ (n°4) correspondant à la date et à l'heure de la requête en utilisant la fonction qui découpe automatiquement la chaîne selon un caractère donné.

Question 3

Ecrire une méthode **get_StatutRequete** qui renvoie sous la forme d'un entier le champ (n°6) correspondant au statut de traitement de la requête par le serveur. Une manière de faire consiste à remarquer que le découpage de la chaîne selon les guillemets permet de trouver le champ cherché en première position du 3e segment.

Question 4

Analyser la chaîne retournée par la question 3 (la date et l'heure de la requête) de façon à isoler sous la forme de 6 entiers : jour, mois, année, heure, minute, seconde. Ces 6 valeurs seront retournées par la méthode **get_FormatedDateRequete** sous la forme d'une seule chaîne (un *timestamp*) répondant au format AAAAMMJJHHMMSS (Année, Mois, Jour, Heure – 24 heures par jour, Minute, Seconde). Le nombre de caractères est important : ainsi la valeur « 1h08 du matin » sera représentée par la séquence 010800. C'est indispensable afin ensuite de pouvoir comparer deux dates sous ce format.

Question 5

Dans le champ n° 5 il y a en réalité trois informations : l'action réalisée (GET, POST, HEAD, etc.), la ressource demandée (/ pour la racine, un nom de fichier, etc.) et le protocole selon lequel la demande a été réalisée (ici c'est HTTP dans sa version 1.1). En cas d'attaque de ver informatique ou de *rootkits* la ressource est « bizarre ». Vous pourrez voir cela dans le fichier log, les premières lignes contiennent la chaîne « /scripts/nsiislog.dll ». C'est une vulnérabilité connue sous certains serveurs Windows exécutant le serveur IIS. On pourrait imaginer une méthode qui renverrait true si la requête (une partie du champ n°5 en particulier) contient une chaîne précise qui serait la signature d'une tentative d'attaque. Ecrire la méthode **is_attack** qui d'après la chaîne complète renvoie *true* si c'est une attaque et *false* sinon. On ne testera dans la chaîne que la présence (et pas l'égalité car il peut y avoir d'autres choses) de « /scripts/nsiislog.dll ».

Modélisation objet

En programmation objet on analyse d'après les fonctionnalités (orienté utilisateur) et on conçoit prioritairement par rapport aux données (orienté réutilisation). Dans le cas présent l'entrée est une chaîne, et la sortie un ensemble de champs caractérisant une requête traitée par le serveur. La valeur ajoutée se situe dans l'interprétation de ces champs et la mise à disposition sous une forme directement accessible de telle ou telle information. La transformation qui est faite pourrait donc être notée : String → { champs }. On va donc chercher à construire un objet qui réalise ces transformations.

Si on s'intéresse au log du site web, on parle de concepts tels que le fichier log ou la requête. La requête fait partie des éléments qui identifient le domaine de l'application, à ce titre on parle d'élément « métier ». En la modélisant sous la forme d'un objet, dont le rôle sera de mettre à disposition « son » contenu, on créera un objet métier, réutilisable dans toutes les applications dont le besoin se situe au niveau de l'interprétation de lignes de log d'un serveur web.

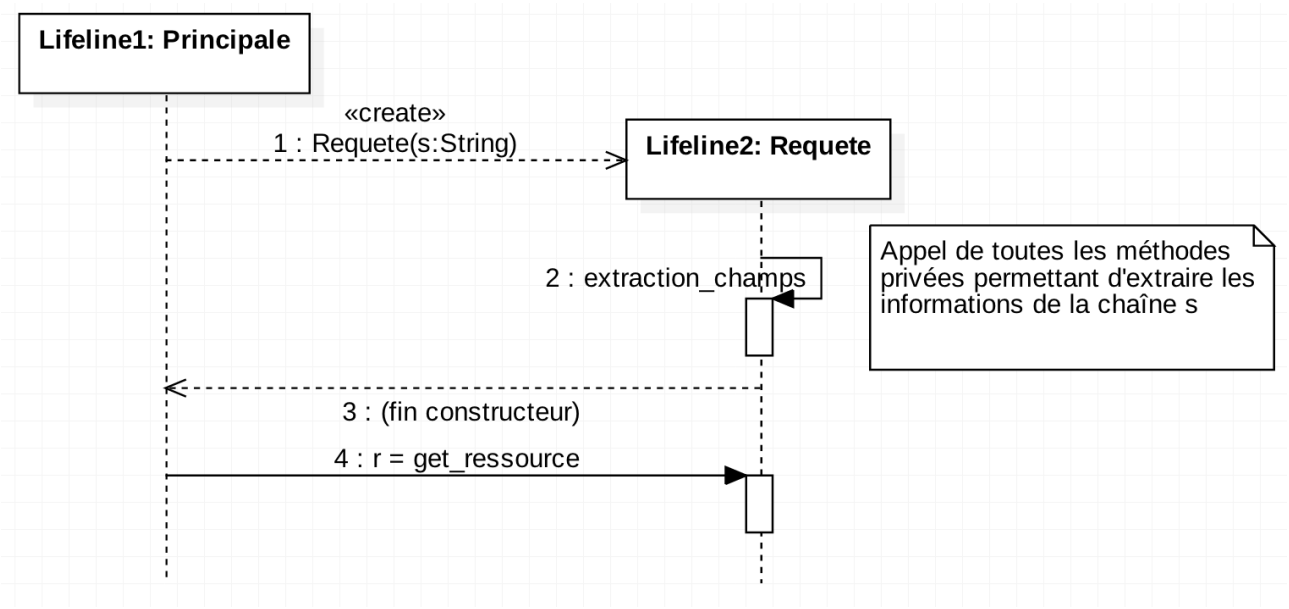
Cet objet qui représente la requête DOIT être opérationnel dès sa création, sans quoi il faudrait prévoir un mécanisme signalant à l'utilisateur que la requête n'est pas initialisée et qu'en conséquence sa demande ne peut aboutir. La création de cet objet doit donc se faire en précisant la chaîne à traiter. Ce constructeur sous-traite à autant de méthode internes (privées donc) que de champs à extraire le découpage correct des informations et leur mémorisation dans des champs (privés eux aussi) de l'objet. Enfin, des fonctions de type « getters » (publiques !) présentent l'information aux utilisateurs.

En résumé, dès qu'on a une chaîne extraite du fichier log, on la donne à un objet de la classe *Requete* lors de son instanciation avec *new*. Le constructeur de la classe (une méthode qui s'appelle comme la classe) fait appel à des méthodes privées qui vont découper la chaîne pour en extraire les champs, les valeurs étant mémorisées dans les champs privés de l'objet instancié. Une fois la découpe terminée, le constructeur se termine. On pourra ensuite, en utilisant les *getters*, demander certaines caractéristiques de cette requête.

La classe rassemble données et méthodes. Ici on a choisi de faire traiter la requête dès le constructeur afin de recopier certaines données dans des champs. Il est évident que deux requêtes différentes auront deux ensembles de champs identiques en noms mais différents en contenu. Chaque objet doit donc avoir ses propres champs. Les champs ne peuvent donc pas être statiques (sinon, tous les objets donc toutes les requêtes, partageraient la même valeur). Et en conséquence, les méthodes ne peuvent pas être statiques (une méthode statique n'accède qu'à des attributs statiques).

On aurait aussi pu imaginer un lot de méthodes qui, recevant la requête, renvoient chacune un des composants. Dans ce cas l'objet ne représente plus une requête mais une bibliothèque de traitements des requêtes enregistrées dans les logs. La transformation agit directement sur l'entrée pour produire la sortie, sans mémorisation intermédiaire. Dans ce cas les méthodes peuvent être statiques et on n'a pas besoin d'attributs.

On choisira le premier point de vie. En UML on représente l'utilisation de cet objet, dans le cas général, sous la forme d'un diagramme de séquences. Par exemple :



Un objet de la classe *Principale* crée un objet de la classe *Requete* en passant à son constructeur la chaîne *s* (autrement dit, on a `Requete req = new Requete(s)` quelque part dans le code de la classe *Principale*. Durant cet appel, des méthodes privées de *Requete* extraient les champs. Puis la classe Principale demande à lire le champ *ressource* (soit `String r = req.get_ressource()`).

On pourra par exemple se référer, pour la structure de la classe, à la représentation ci-dessous (à gauche UML, à droite Java) pour programmer la partie B. On s'appuiera pour cela sur les traitements de base. Ne sont représentées que les attributs privés et les méthodes publiques. Il faut rajouter les méthodes privées. Un exemple de *getter* est donné (en bleu)

Question 6

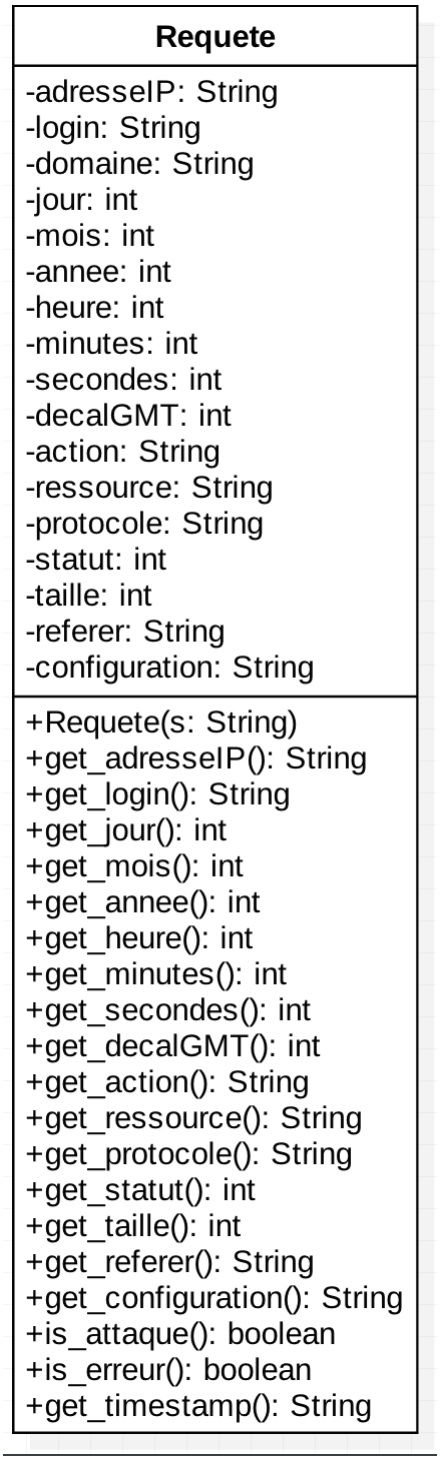
Programmez la classe *Requete* et faites quelques essais de lecture des champs depuis votre classe principale. Vous pouvez ne faire que quelques méthodes, à condition d'avoir bien compris le mécanisme d'appel entre constructeur, méthode privées d'extraction (qui sont des *setters*) et *getters*.

Question 7

Réalisez un petit programme de démo qui montre le fonctionnement de l'objet Requête.

RAPPEL : en fait on fait le contraire. On commence par écrire le programme de démo, et on travaille ensuite la programmation de la classe Requête jusqu'à ce que le programme de démo soit satisfait.

De la même manière, on commence par écrire la documentation (javadoc) d'une méthode en expliquant ce qu'elle fait, accepte en entrée, produit en sortie. Dans le corps de la méthode on place les principales étapes là-aussi sous forme de commentaires de façon à (s') expliquer comment cela fonctionne. Et après, quand c'est clair, on peut coder. Une difficulté à la fois.



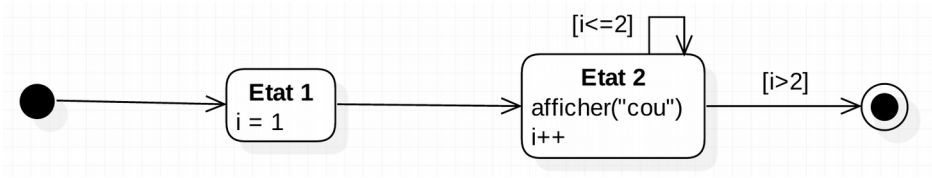

```
public class Requete {

    private String adresseIP;
    private String login;
    private String domaine;
    private int jour;
    private int mois;
    private int annee;
    private int heure;
    private int minutes;
    private int secondes;
    private int decalGMT;
    private String action;
    private String ressource;
    private String protocole;
    private int statut;
    private int taille;
    private String referer;
    private String configuration;


    public Requete(String s) {}
    public String get_adresseIP() {}
    public String get_login() {}
    public int get_jour() {}
    public int get_mois() {return mois ;}
    public int get_annee() {}
    public int get_heure() {}
    public int get_minutes() {}
    public int get_secondes() {}
    public int get_decalGMT() {}
    public String get_action() {}
    public String get_ressource() {}
    public String get_protocole() {}
    public int get_statut() {}
    public int get_taille() {}
    public String get_referer() {}
    public String get_configuration() {}
    public boolean is_attaque() {}
    public boolean is_erreur() {}
    public String get_timestamp() {}
}
```

Approfondissement

On suppose l'existence d'un automate capable de prendre différents états en fonction d'évènements constatés. Au début il y a un état imposé (l'état *initial*). Puis, en fonction de différents événements ou conditions, l'automate va évoluer, changer d'état. Certains états ne permettent plus à l'automate d'évoluer, on les appellera état *finaux*. Petit exemple :

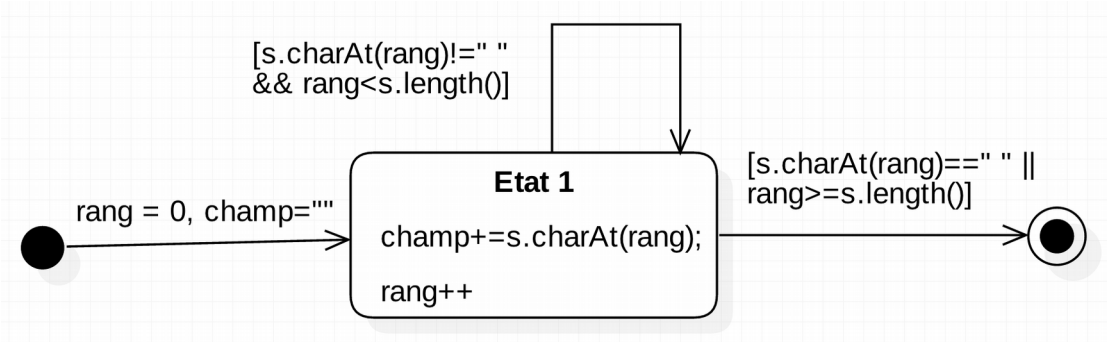


L'état initial est désigné comme étant l'état 1. Ce dernier ne réalise qu'une seule action, affecter la valeur 1 à la variable i. Puis une transition sans condition (donc toujours réalisée) « nous emmène » dans l'état 2. Là deux actions sont réalisées : afficher (« cou ») d'une part, et incrémenter i de 1 d'autre part. Deux transitions sont présentes : l'une est gardée par la condition [i<=2] et l'autre par la condition [i>2]. Il est VITAL à cet instant que seule une condition soit vraie, de façon à ce qu'on sache assurément vers quel état on va aller (sans quoi la machine n'est pas déterministe et dans ce cas on n'accepte pas ce mode de fonctionnement). Dans le cas [i<=2] on retourne dans l'état 2, dans l'autre cas on va vers un état final (cela marque la fin de l'automate). De retour dans l'état 2 on refait les mêmes action : afficher (« cou ») puis incrémenter i (qui maintenant vaut 3). La condition [i>2] est maintenant vraie, ce qui nous emmène vers l'état final. La machine aura donc affiché « coucou ».

Ce type de représentation est facile à programmer, d'ailleurs il existe des logiciels qui produisent des programmes d'après des représentations de ce genre (c'est le cas de AnyLogic de XJ Technologies). C'est aussi le formalisme qui permet de représenter l'évolution des état d'un objet en UML (diagramme d'état). On utilise une structure de type «switch» afin d'énumérer les actions des différents états. Une variable représente le numéro de l'état courant. Cette variable est mise à jour à l'intérieur des blocs du switch. Un « while » englobe le tout et vérifie la valeur d'une variable qui est modifiée uniquement dans les états finaux. L'automate ci-dessus se traduit en java par :

```
int etat = 1;
boolean encore = true;
int i = 0;
while (encore){
  switch (etat) {
    case 1:
      i = 1;
      etat = 2;
      break;
    case 2:
      System.out.print("cou");
      i++;
      if (i <= 2) etat = 2;
      else if (i > 2) etat = 3;
      break;
    case 3:
      encore = false;
      break;
  }
}
```

Dans le même esprit, l'analyseur de chaîne se conçoit comme un ensemble de groupes d'états, chaque groupe étant chargé de « reconnaître » l'un des champs. Une variable désigne le rang du caractère traité. Ainsi, pour le champ n°1 (adresse IP) on aura la logique suivante :



Avant d'entrer dans l'état 1 on dispose de deux variables (*rang* qui vaut 0 et *champ* qui contient la chaîne vide). Une fois dans l'état 1 on concatène à la chaîne *champ* le caractère de rang *rang* de la chaîne *s* et on incrémente le rang. Selon les cas on peut reboucler sur cet état (tant que le caractère considéré est différent d'un espace et que le rang existe) ou aller vers l'état de fin (dès que le caractère considéré est un espace ou bien que le rang est trop grand pour indexer un caractère de *s*.

On crée ensuite un second automate qui va extraire de la même manière le second champ, puis on relira l'état initial de ce second automate à l'état final de l'automate précédent. Ainsi, on passera du dernier état actif de l'automate 1 au premier état actif de l'automate 2. Et ainsi de suite. On aboutit à un automate plus ou moins complexe, selon la quantité de traitements modélisés.

Question 8

Finir la modélisation sous la forme d'un automate de façon à représenter l'extraction des champs.

Question 9

Programmer l'automate de la question 8 et l'intégrer dans la classe `Requete`. Deux options : supprimer les méthodes privées et placer l'automate dans le constructeur ; ou bien découper l'automate et placer chaque morceau dans les méthodes privées. Dans ce dernier cas il faut essayer de faire en sorte que chaque méthode privée retourne la portion de chaîne qu'elle n'a pas traitée. Ce résultat est donnée à la méthode suivante comme point de départ (en effet, chaque méthode faisant appel à un automate spécialisé qui suppose être au début du champ....).

Vous pouvez aussi aller voir sur le cours en ligne, l'exercice supplémentaire n°7 : 4-EAP-7 - Machine de Turing

Annexe 1 : Format typique d'une ligne de fichier log (serveur HTTP Apache)

```
195.242.73.130 - - [23/Oct/2003:10:26:48 +0200] "GET / HTTP/1.1" 200 324 "-" "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98; Win 9x 4.90)"
```

Position	Délimiteur	Valeur dans l'exemple	Signification
1	espace	`195.242.73.130`	Adresse IP du client à l'origine de la requête
2	espace	`-`	Dans un réseau authentifié, login de l'utilisateur
3	espace	`-`	Dans un réseau authentifié, domaine de connexion
4	`[ ]`	`23/Oct/2003:10:26:48 +0200`	Date et heure de la requête (jour/mois/année:heure:minutes:seconde décalageGMT)
5	`" "`	`GET / HTTP/1.1`	Requête adressée au serveur
6	espace	`200`	Statut de traitement de la requête (cf. RFC 2616[1])
7	espace	`324`	Taille de la réponse retournée au client, en octets
8	`" "`	`-`	URL de la page référente : celle dans laquelle on a cliqué pour émettre la requête en cours
9	`" "`	`Mozilla/4.0 (compatible; MSIE 5.5; Windows 98; Win 9x 4.90)`	Identification du navigateur émetteur de la requête

1 Par exemple à http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html