CSC3101 - CF2 - La Gestion du Contexte dans les Systèmes d'IA

24 mars 2026

Prénom Nom: _____

Lisez attentivement les instructions données ci-dessous, et vérifiez que vous avez bien toutes les pages (10 au total). Cet examen papier dure une heure et demie, et les documents sont autorisés. L'usage d'un appareil connecté (ordinateur, tablette, etc.) est interdit. Soyez sûr que votre réponse est lisible. Cet examen est noté sur 20 points. N'hésitez pas à sauter des questions si vous êtes bloqués. Vous pouvez répondre aux questions suivantes en utilisant les fonctions supposées implémentées aux questions précédentes (pensez à regarder les signatures).

Quand la visibilité n'est pas indiquée, on pourra laisser celle par défaut. On suppose que toutes les classes sont dans le même package, qui ne doit pas être indiqué.

Introduction

Les modèles de langage modernes, tels que ceux alimentant les agents conversationnels, ne sont pas de simples générateurs de texte isolés. Ils s'inscrivent au sein de systèmes complexes chargés de maintenir la cohérence d'un dialogue sur le long terme. Une contrainte technique majeure de ces modèles est leur **fenêtre de contexte** : ils ne peuvent traiter qu'une quantité limitée d'informations à un instant donné. Si une conversation devient trop longue, le système doit être capable de "nettoyer" l'historique sans pour autant perdre les instructions fondamentales qui dictent son comportement.

Dans cet examen, vous allez concevoir un *SDK* (Software Development Kit) simplifié dédié à la gestion de ces contextes de dialogue. Le défi ne réside pas seulement dans l'algorithmique, mais dans la mise en place d'une architecture logicielle robuste et évolutive. Vous devrez modéliser les différents types d'intervenants (Utilisateur, Assistant, Système) et garantir l'intégrité des échanges grâce aux principes de la Programmation Orientée Objet.

La première partie de l'examen portera sur la modélisation des messages via l'**héritage** et l'**encapsulation**, ainsi que sur la sécurisation des données par le biais des **exceptions**. Vous devrez notamment vous assurer que les messages respectent des contraintes de formatage et de taille avant d'être transmis au modèle.

Ensuite, vous explorerez l'utilisation de la **généricité** pour manipuler des réponses de types variés, permettant ainsi à votre système de s'adapter à différents formats de sortie (texte brut, scores numériques, ou métadonnées). Enfin, vous implémenterez un algorithme de **Context Trimming** (élagage de contexte). Cet algorithme devra gérer intelligemment une liste dynamique de messages afin de respecter les limites de mémoire tout en préservant impérativement les instructions de base du système.

Le but de cet examen est de vous confronter à une problématique réelle de l'ingénierie de l'IA : comment structurer et maintenir un état cohérent dans un environnement contraint, tout en appliquant les concepts fondamentaux de la programmation Java.

1 Modélisation et Robustesse des Échanges (6,5 points)

Dans un système de dialogue, chaque interaction est représentée par un message. Cependant, tous les messages n'ont pas le même rôle. On distingue généralement trois types :

- Le **System Message** : Il définit le comportement de l'IA (ex: "Tu es un traducteur"). Il est unique et définit les consignes de base.
- Le **User Message** : Il contient la requête envoyée par l'utilisateur humain.
- Le **Assistant Message** : Il contient la réponse générée par l'intelligence artificielle.

1. (1 point) Définissez une exception personnalisée nommée *InvalidMessageException*. Elle devra posséder un constructeur prenant en entrée une chaîne de caractères *message* et appelant le constructeur de la classe parente avec cette même chaîne en entrée.

Solution:

```
1 public class InvalidMessageException extends Exception {
2     public InvalidMessageException(String message) {
3         super(message);
4     }
5 }
```

2. (2 points) Créez une classe abstraite nommée *Message*. Cette classe doit :
   - Posséder un attribut privé *content* (String).
   - Posséder un constructeur qui initialise cet attribut. Si le contenu passé en paramètre est `null` ou vide (`""`), le constructeur doit lever une *InvalidMessageException* avec comme message "Contenu invalide".
   - Posséder une méthode getter publique *String getContent()* qui retourne *content*.
   - Posséder une méthode abstraite publique *String format()*.

Solution:

```
1  public abstract class Message {
2      private String content;
3
4      public Message(String content) throws InvalidMessageException {
5          if (content == null || content.isEmpty()) {
6              throw new InvalidMessageException("Contenu invalide");
7          }
8          this.content = content;
9      }
10
11     public String getContent() { return content; }
12     public abstract String format();
13 }
```

3. (0.5 points) Peut-on exécuter l'instruction suivante : `Message m = new Message("Hello");` ? Justifiez.

Solution: Non. La classe *Message* est **abstraite**, elle ne peut donc pas être instanciée directement.

4. (1 point) Pourquoi est-il préférable de déclarer l'attribut *content* comme `private` ? Quel concept de la programmation orientée objet cela illustre-t-il ?

Solution: Cela illustre l'**encapsulation**. Cela permet de protéger les données internes de l'objet et d'empêcher une modification sauvage qui contournerait les règles de validation définies dans le constructeur.

5. (2 points) Implémentez les classes *UserMessage* et *SystemMessage* qui héritent de *Message*.

   - Chaque classe doit implémenter la méthode *String format()* qui retourne le contenu préfixé par le rôle (ex: `"[USER]: ..."` ou `"[SYSTEM]: ..."`).
   - Assurez-vous de gérer correctement l'appel au constructeur de la classe parente.

Solution:

```
1  public class UserMessage extends Message {
2      public UserMessage(String content) throws InvalidMessageException {
3          super(content);
4      }
5      public String format() {
6          return "[USER] : " + getContent();
7      }
8  }
9
10 public class SystemMessage extends Message {
11     public SystemMessage(String content) throws InvalidMessageException {
12         super(content);
13     }
14     public String format() {
15         return "[SYSTEM] : " + getContent();
16     }
17 }
```

2 Formatage des Réponses (3,5 points)

Les modèles de langage ne retournent pas uniquement du texte brut. Selon la tâche demandée, ils peuvent renvoyer un score numérique (ex: analyse de sentiment), une probabilité, ou même une liste de mots-clés. Pour gérer cette diversité, nous allons utiliser la généricité.

1. (1.5 points) Créez une classe générique nommée *ModelResponse* prenant un paramètre de type T. Cette classe doit :
   - Posséder deux attributs privés : *content* (de type T) et *modelName* (de type *String*).
   - Posséder un constructeur prenant en entrée ces deux paramètres pour initialiser les attributs.

Solution:

```
1  public class ModelResponse<T> {
2      private T content;
3      private String modelName;
4
5      public ModelResponse(T content, String modelName) {
6          this.content = content;
7          this.modelName = modelName;
8      }
9
10     public T getContent() { return content; } // non demande
11 }
```

2. (1.5 points) On souhaite utiliser cette classe dans un programme principal (*main*). Écrivez une méthode main dans la classe *Main* (on ne demande pas de créer cette classe) avec les lignes de code créant les deux objets suivants :
   1. Une réponse nommée *resp1* contenant le texte `"Bonjour"` provenant du modèle `"TéléGPT"`
   2. Une réponse nommée *resp2* contenant la valeur entière `5` (score de satisfaction) provenant du modèle `"TéléDall-E"`.

Solution:

```
1 public static void main(String[] args){
2     ModelResponse<String> resp1 = new ModelResponse<>("Bonjour", "TeleGPT");
3     ModelResponse<Integer> resp2 = new ModelResponse<>(5, "TeleDall-E");
4 }
```

3. (0.5 points) Quel est l'avantage principal d'avoir utilisé un type générique T pour l'attribut *content* plutôt que le type *Object* ?

Solution: Cela permet de détecter les erreurs à la compilation plutôt qu'à l'exécution et évite d'avoir à effectuer des conversions de type (casts) explicites et risquées lors de la récupération du contenu.

3 Gestion de la Conversation et Élagage (10 points)

La classe *Conversation* est le cœur du système. Elle stocke l'historique des échanges sous forme de liste et s'assure que la taille totale ne dépasse pas les capacités du modèle (la taille du contexte, c'est-à-dire la taille maximale d'un texte pouvant être lu par notre IA), tout en préservant les instructions initiales.

Une conversation commence toujours par un message système *SystemMessage unique* qui ne doit jamais être supprimé, puis continue avec une séquence de messages de l'utilisateur *UserMessage* ou de l'assistant *AssistantMessage* (n'a pas été demandé dans l'énoncé) dans l'ordre dans lequel ils ont été écrits. Les messages de l'utilisateur sont susceptibles d'être supprimés si la conversation devient trop longue, en commençant par les plus anciens.

Rappel utile : La classe `ArrayList<E>` *de la bibliothèque* `java.util` *met à disposition les méthodes suivantes :* `add(E e)`, `remove(int index)`, `size()` *et* `get(int index)`.

1. (1.5 points) Créez la classe *Conversation*. Elle doit posséder un attribut privé nommé *history* qui est une liste dynamique d'objets de type *Message*. Pour l'attribut *history*, vous prendrez soin d'utiliser un type

de liste qui fonctionne quelle que soit l'implémentation sous-jacente (aide : utilisez une interface fournie par Java). Initialisez cette liste dans le constructeur avec une *ArrayList* vide. Les *imports* sont considérés comme optionnels pour cette question.

Solution:

```
1 public class Conversation {
2     private List<Message> history;
3
4     public Conversation() {
5         this.history = new ArrayList<>();
6     }
7 }
```

2. (0.5 points) Implémentez une méthode *void addMessage(Message m)* qui ajoute un message à l'historique.

Solution:

```
1 public void addMessage(Message m) {
2     this.history.add(m);
3 }
```

3. (2 points) Implémentez une méthode *int calculateTotalChars()* qui calcule et retourne la somme du nombre de caractères de tous les messages présents dans l'historique.

Solution:

```
1 public int calculateTotalChars() {
2     int total = 0;
3     for (Message m : history) {
4         total += m.getContent().length();
5     }
6     return total;
7 }
```

4. (1 point) Quelle est la complexité temporelle de la méthode *calculateTotalChars()* en fonction du nombre de messages N présents dans l'historique ? Justifiez brièvement.

Solution: La complexité est $\mathcal{O}(N)$. Justification : On parcourt une seule fois la liste des N messages.

5. (2 points) Implémentez la méthode *void trimContext(int maxChars)* qui réduit la taille de l'historique tant que le nombre total de caractères est supérieur à *maxChars*. **Règles :**

   - Le premier message (index 0) ne doit **jamais** être supprimé.
   - On supprime les messages les plus anciens (situés juste après le message système).

- La suppression s'arrête s'il ne reste qu'un seul message dans la liste.

Solution:

```
1 public void trimContext(int maxChars) {
2     while (calculateTotalChars() > maxChars && history.size() > 1) {
3         history.remove(1);
4     }
5 }
```

6. (1 point) Analysez la complexité temporelle de la méthode *trimContext(int maxChars)* dans le pire des cas, en fonction de N, la taille de *history*. On rappelle que supprimer un élément dans une *ArrayList* implique de décaler de un vers la gauche tous les éléments suivants.

Solution: La complexité est $\mathcal{O}(N^2)$. Justification : On peut effectuer jusqu'à N itérations, et à chaque fois `calculateTotalChars()` (en fonction de l'implémentation, en $\mathcal{O}(N)$) et `remove(1)` (en $\mathcal{O}(N)$) sont appelées.

7. (1 point) Pour éviter le coût du décalage des éléments lors d'une suppression au début de la liste, quelle autre structure de données de la bibliothèque `java.util` serait plus efficace qu'une *ArrayList* ? Justifiez.

Solution: Une **LinkedList** (Liste chaînée), car la suppression en tête (ou proche de la tête) se fait en temps constant $\mathcal{O}(1)$ sans décalage mémoire.

8. (1 point) Implémentez une méthode *boolean containsKeyword(String keyword)* qui retourne vrai si au moins un message de l'historique vaut exactement le texte *keyword*. Quelle méthode de String devez-vous utiliser pour comparer les contenus ?

Solution:

```
1 public boolean containsKeyword(String keyword) {
2     for (Message m : history) {
3         if (m.getContent().equals(keyword)) return true;
4     }
5     return false;
6 }
```

On utilise la méthode `.equals()` (et non `==`).