Epreuve écrite Conception Orienté Objet
SI4 – 6 décembre 2021 – 3 h

Le total du barème est de 24 points. Lisez soigneusement les énoncés et choisissiez les questions auxquels vous répondrez. Durant l'examen, nous ne répondrons pas aux questions orales sur l'interprétation du sujet. Si vous avez un doute, décrivez-le sur votre copie.

Exercice Modélisation (8 points)

Vous devez modéliser un système de demande de courses pour une compagnie de véhicules avec chauffeur (type VTC). Une course est un parcours demandé par un client entre une adresse de départ et une adresse de destination. Le système est composé de deux applications mobiles (une application pour les clients et une pour les chauffeurs) et d'un backend serveur. **Vous ne modéliserez que la partie serveur.**

Les endroits où la société opère sont définis en zones géographiques (e.g. Paris, Nice, New York, ...). Ces zones sont associées à un ensemble de coordonnées géographiques (contour de la zone), une devise (€, $, £, ...), un tarif kilométrique, un tarif de prise en charge, un tarif horaire, et un tarif minimum.
Les chauffeurs sont présents dans une zone géographique. Ils ont un ou plusieurs véhicules (identifiés par marque, modèle, couleur, immatriculation). Ils s'authentifient sur l'application avec un numéro de téléphone et un mot de passe. Ils peuvent sélectionner le véhicule qu'ils utilisent ou en enregistrer un nouveau. Ils quittent explicitement l'application lorsqu'ils ont fini leur journée de travail. Lorsque l'application est en route, elle envoie toutes les minutes la position GPS du chauffeur au système.
Les clients s'authentifient grâce à un numéro de téléphone et un mot de passe. Au moment de leur enregistrement dans l'application leur numéro de téléphone est vérifié par l'envoi d'un code par SMS et ils doivent saisir un numéro de carte visa. Les clients peuvent demander une course pour un départ immédiat ou pour une date ultérieure (réservation). Le système vérifie que les adresses de départ et de destination sont bien dans la zone géographique. Pour trouver les coordonnées géographiques d'une adresse on utilise l'API Google Map.
Lorsqu'un client demande une course immédiate ou 15 minutes avant le départ d'une course réservée, le système identifie dans la zone géographique les 10 chauffeurs libres les plus proches (à vol d'oiseau), puis leur demande (dans l'ordre) s'ils acceptent la course. Si aucun n'accepte la requête est relancée 5min plus tard. Le client est notifié lorsqu'un chauffeur accepte la course. Le client peut annuler sa demande si aucun chauffeur n'a encore accepté. Le client voit la position de son chauffeur sur son application depuis l'acceptation de la course jusqu'à la fin de celle-ci. A la fin de la course, l'application du chauffeur envoie la distance et le temps passé, le prix de la course est calculé puis facturé automatiquement au client (en utilisant un service externe bancaire) et un SMS lui est envoyé.

Faire les diagrammes suivants :

- [1 pt] Diagramme de cas d'utilisation
- [3 pt] Diagramme de classes
- [1 pt] Diagramme de séquences de l'ajout d'un véhicule par un chauffeur

- [3 pt] Diagramme de séquences depuis la demande d'une course immédiate jusqu'à la prise en charge (monté du client dans le véhicule).

Exercice Gherkin (8 points)

Soit les squelettes de classes suivants représentant un service de transactions bancaires.

```
@Service
public class TransactionService {
    // Finds a specific transaction
    public Optional<Transaction> findById(UUID reference);
    // Registers a transaction
    public void save(Transaction transaction);
    // Finds transaction by IBAN
    public List<Transaction> findByIban(String accountIban);
    // Finds a specific transaction status according transaction UUID
    public TransactionStatus getTransactionStatus(UUID reference) {
}
```

```
@Entity
public class Transaction {
    private UUID reference;
    private String accountIban;
    private ZonedDateTime date;
    private float amount;
    private float fee;
    private String description;

    public Transaction();
    public Transaction(UUID reference, String accountIban, ZonedDateTime date, float amount,
                       float fee, String description)
}
```

```
@Entity
public class TransactionStatus {
    private UUID reference;
    private Status status;
    private float amount;
    private fee fee;
}

public enum Status {
    PENDING, SETTLED, FUTURE, INVALID
}
```

Notes :

- L'annotation **@Entity** vous permet d'avoir les accesseurs correspondants à vos attributs privés (e.g. getReference())

```
public class transactionServiceStepDefTest {
    private static final UUID TRANSACTION_REFERENCE = UUID.randomUUID();
    private static final String ACCOUNT_IBAN = "ES9820385778983000760236";

    // Injection d'une instance de TransactionService
    @Autowired
    private TransactionService transactionService;
```

✓ **Ecrire un fichier feature [2 pt] et complétez la classe ci-dessus pour écrire les step defs correspondants aux scénarios suivants :**

1) [2 pt] Si l'on enregistre une transaction invalide (valeur accountIban = "XX"), son statut est INVALID
2) [2 pt] Si l'on enregistre une transaction valide avec une date après la date actuelle, son statut est FUTURE
3) [2 pt] Si l'on enregistre deux transactions sur un même IBAN la méthode de recherche les retrouve

Exercice Conception (8 points)

Soit un micro language de programmation avec une grammaire non-contextuelle - analyseur LR(1) - qui ressemble à cela :

```
x := 7;
tantque x > 0 {
  si x % 2 = 0 alors n := n * x;
  x := x - 1;
}
```

Voici le squelette du code permettant l'analyse et l'interprétation de ce language. Pour chaque classe on estime qu'elle contient les accesseurs et modificateur pour tous ses attributs et qu'elle implémentente toutes les méthodes provenant de leur interface.

```
// An instruction
public interface Instruction {
  // Evaluates the instruction in a given context (may modify the context)
  void evaluate(Context context);
}

// An ordered block of instructions
public class InstructionBlock implements Instruction {
  private List<Instruction> instructions;
}

// If Instruction
public class IfInstruction implements Instruction {
  private Expression condition;
  private Instruction thenInstruction;
  private Instruction elseInstruction;
}

// While Instruction
public class WhileInstruction implements Instruction {
  private Expression condition;
  private Instruction bodyInstruction;
}

// Assign Instruction
public class AssignInstruction implements Instruction {
  private Reference leftPart;
  private Expression rightPart;
}
```

```
public enum Type {...}
public enum Operator {...}

// A typed value
public class Value {
  private Object value;
  private Type type;
}

// Associates variable name to values
public class Context {
  private Map<String, Value> values;
}

// An expression
public interface Expression {
  // Evaluates the expression in a given context
  Value evaluate(Context context);
}

// A literal (e.g. 42 or "Hello")
public class Literal implements Expression {
  private Value value;
}

// A variable
public class Reference implements Expression {
  private String name;
}

// An operation
public class Operation implements Expression  {
  private Expression leftOperand;
  private Expression leftOperand;
  private Operator operator;
}
```

```
// Reads tokens on an InputStream
public class TokenReader {
  public TokenReader(InputStream in)
  public String nextToken();
}

public class Expressionyser {
  public Expressionyser();
  // Parses a literal
  public Expressionyser literal();
  // Parses a variable name
  public Expressionyser reference(String name);
  // Parses an operation
  public Expressionyser operation(Operator operator);
  // Returns the expression
  public Expression result() throws MalformedExpressionException;
}

public class Analyser {
  public static Instruction analyse(InputStream in) throws MalformedInstructionException;
  public static Context evaluate(Instruction instruction);

  private static Instruction parse(TokenReader tr) throws MalformedInstructionException;
  private static IfInstruction parseIf(TokenReader tr) throws MalformedInstructionException;
  private static WhileInstruction parseWhile(TokenReader tr) throws MalformedInstructionException;
  private static AssignInstruction parseAssign(TokenReader tr) throws MalformedInstructionException;
}
```

Questions :

1) [4 pt] Quels patrons de conception sont présents dans le code ci-dessus ? Justifiez en indiquant quelle classe du code correspond à quelle classe dans le patron.
2) Pour les fonctionnalités suivantes, indiquez quel(s) patron(s) de conception serait le mieux adapté ? Justifiez en quelques lignes et indiquez, si besoin, les modifications à apporter aux classes exitantes (méthodes ou attributs à modifier/ajouter).
   a) [2 pt] Une génération de code depuis l'analyse de ce language vers plusieurs languages de programmation structuré (python, js, ruby, ...).
   b) [2 pt] En plus on veut aussi pouvoir générer du code vers diverses languages types assembleurs. Pour cela il faut pouvoir associer chaque expression à un registre dont le nom est formé différemment suivant le language cible. Par exemple, en X86 ((10+20)+30) devient :

```
const/16 v0, #int 10
const/16 v1, #int 20
add-int v2, v0, v1
const/16 v3, #int 30
add-int v4, v2, v3
```