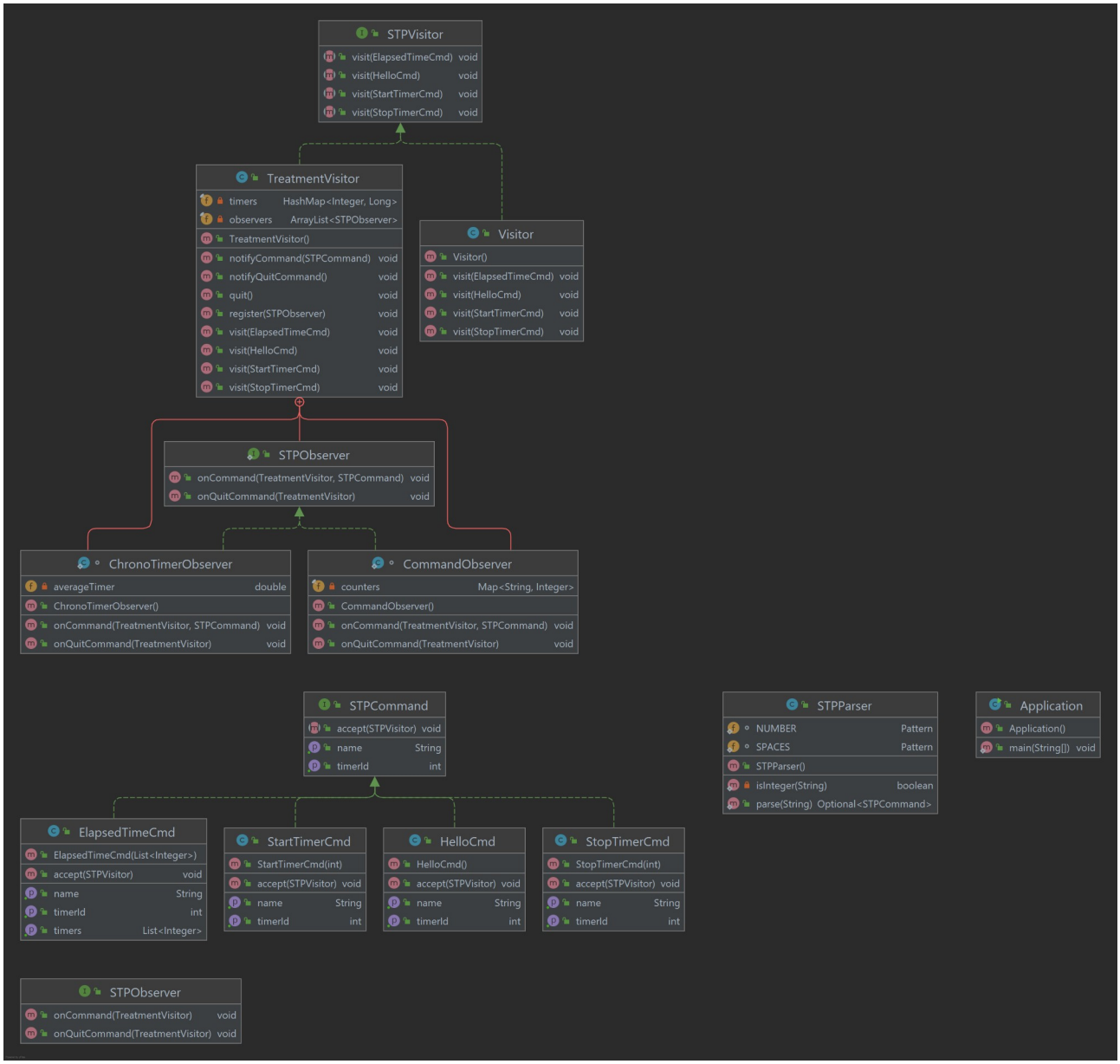
Rapport Visitor

Table des matières

- **Exercice 1**



Tout d'abord nous avons implémenté le patron Visitor qui nous permet d'ajouter des fonctionnalités tel que start, stop, hello, elpased. Sans avoir besoin de modifier l'interface que ces fonctionnalités implémentent ni le code des classes. Grâce au patron Visitor on a plus besoin de faire des instanceof pour vérifier le type si la commande est un hello, start… En effet ceci est maintenant réalisé grâce à la méthode accept qui prend en paramètre un STPVisitor de l'interface STPCommand implémenté par chaque classe représentant une commande.

Afin d'implémenter les fonctionnalités tel que le calcule du nombre de commandes de chaque type et calcule le temps moyen entre le démarrage et l'arrêt d'un chronomètre nous avons utilisé le patron Observer. Tout d'abord nous implémentons une interface contenant les méthode pour l'exécution d'une commande et une méthode lorsque l'on quitte le programme. Il faut ensuite créer

des classes correspondante à ces fonctionnalité et qui implémente l'interface Observer. Pour cela on a décidé de les implémenté en classe interne dans la classe TreatmentVisitor directement ainsi que la liste d'observers et les méthode qui notify lorsqu'une commande est saisie ou lorsque l'on quitte le programme. De plus une méthode quit() a été rajouté afin de notifier lorsque l'on quitte le programme et ainsi affiché les données collectés pour chaque fonctionnalités. Pour la fonctionnalité qui enregistre le nombre de commande réalisé par type, il a fallu créer une méthode getName() dans l'interface STPCommand afin de savoir quelle commande à été faite.

```
static class CommandObserver implements STPObserver {
    private final Map<String, Integer> counters = new HashMap<>();

    @Override
    public void onCommand(TreatmentVisitor treatmentVisitor, STPCommand stpCommand) {
        var commandName :String = stpCommand.getName();
        var counter :int = counters.get(commandName) != null ? counters.get(commandName) : 0;
        counters.put(commandName, counter + 1);
    }

    @Override
    public void onQuitCommand(TreatmentVisitor treatmentVisitor) {
        counters.forEach((name, count) -> System.out.println(name + " : " + count));
    }
}
```

A chaque commande on l'enregistre dans une map ayant pour clé le nom de la commande et pour valeur son nombre d'appel puis on l'affiche lorsque l'on quitte le programme. Pour le temps moyen des chrono on incrémente un compteur que l'on divise par la taille de la map timers qui correspond au nombre de chrono lancer puis on l'affiche aussi si on quitte le programme.

```
static class ChronoTimerObserver implements STPObserver {
    private double averageTimer;

    @Override
    public void onCommand(TreatmentVisitor treatmentVisitor, STPCommand stpCommand) {
        var total = 0L;
        if (treatmentVisitor.timers.get(stpCommand.getTimerId()) != null) {
            total = System.currentTimeMillis() - treatmentVisitor.timers.get(stpCommand.getTimerId());
        }
        averageTimer += total;
    }

    @Override
    public void onQuitCommand(TreatmentVisitor treatmentVisitor) {
        averageTimer /= treatmentVisitor.timers.size();
        System.out.println("Average timer : " + averageTimer + "ms");
    }
}
```

- ## Exercice 2-3

Dans le visitor de l’exercice 2 nous étions obligé d’avoir une méthode accept() permettant d’accepter un exprVisitor car on pouvait avoir Value ou BinOp. Tandis que dans le visitor de l’exercice cette méthode n’est plus nécessaire car toute la logique est dans le main que se soit pour le toString ou bien le calcul. Dans l’exercice 3 la méthode when nous permet notamment de chaîner les opérations que se soit pour construire le toString ou bien appliqué le calcul on a donc une seule méthode pour toutes les opérations tandis que dans l’exercice 2 on avait une classe pour toString une autre pour ExprVisitor. On peut donc dire que le visitor de l’exercice 3 permet d’écrire moins de code que le visitor de l’exercice 2.