POCA — TP noté n°1
Java générique

Jean-Baptiste Yunès
`Jean-Baptiste.Yunes@u-paris.fr`

12 octobre 2023

Rendu final

L'intégralité du code produit, les tests, etc doit être fourni à la date indiquée par l'enseignant. Il est important de ne fournir que le code source, et de prendre bien soin que rien ne manque. Le code sera fourni sous la forme d'une archive `tar.gz` ou `zip` (pas autre chose). Le code devra comprendre un fichier `README.txt` contenant le nom et le prénom (dans cet ordre !) des élèves concernés. Les modalités de rendu seront donnés par l'enseignant mais se feront sur `moodle.u-paris.fr` dans le cours POCA et sous aucune autre forme.

Généralités

Dans ce premier TP, on vise à construire un système de distribution d'évènements à l'aide de type génériques

Question n°1

Pour un système donné :

- les évènements sont de type désigné par `T` ;
- fin de recevoir un évènement, un objet doit être du type `EventListener<T>` et dans ce cas implémenter la méthode `accept` (par exemple afficher un message particulier additionné de l'évènement lui-même ou n'importe quel traitement de la donnée de type `code`).
- la distribution s'effectue *via* des instances de la classe `EventDispatcher`. Afin qu'un évènement puisse être reçu par un listener il faut au préalable que celui-ci **s'enregistre** auprès du dispatcher *via* sa méthode `addListener`. La **distribution** d'un évènement à tous les listeners enregistrés s'effectue par appel à la méthode `send` du dispatcher concerné. Enfin, il est possible à un listener de se désabonner par appel à la méthode `removeListener` du dispatcher auprès duquel il s'était

préalablement enregistré.

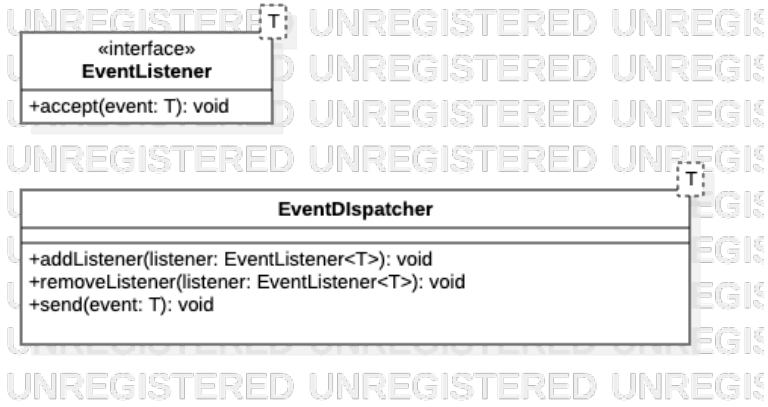

Figure 1 – UML

Implémentez l'interface et la classe décrite (complétez avec n'importe quel champ supplémentaire mais ne pas rajouter de méthode).

Testez des assemblages avec différents type pour l'instanciation des génériques. Pour un assemblage donné, distribuez un évènement à un dispatcher vide, ajoutez un listener, envoyez un évènement, ajoutez un autre listener, envoyez un évènement, supprimez le premier listener, envoyez un évènement, etc.

Question n°2

Modifiez le TP précédent de sorte que l'on puisse faire tous les assemblages raisonnables. Pour vous guider, posez-vous des questions du type :

«puis-je utiliser un `EventListener<Integer>` avec un `EventDispatcher<Number>` ?»

Implémentez, testez.

Note : il faut certainement modifier les paramètres génériques...

Question n°3

Cette fois on enrichi fonctionnellement le système de distribution de sorte que lorsqu'on enregistre, auprès d'un dispatcher, un listener, on lui associe un filtre d'évènement (instance de `EventFilter<T>`), à l'aide d'une surcharge de la méthode `addListener`. Ainsi la distribution à ce listener est conditionnée au prédicat du filtre, i.e. un événement n'est distribué au listener que si le filtre renvoie vrai pour cet événement. L'enregistrement sans filtre associé (la première version de la méthode), correspond maintenant à un enregistrement associé au filtre qui renvoie inconditionnellement vrai.

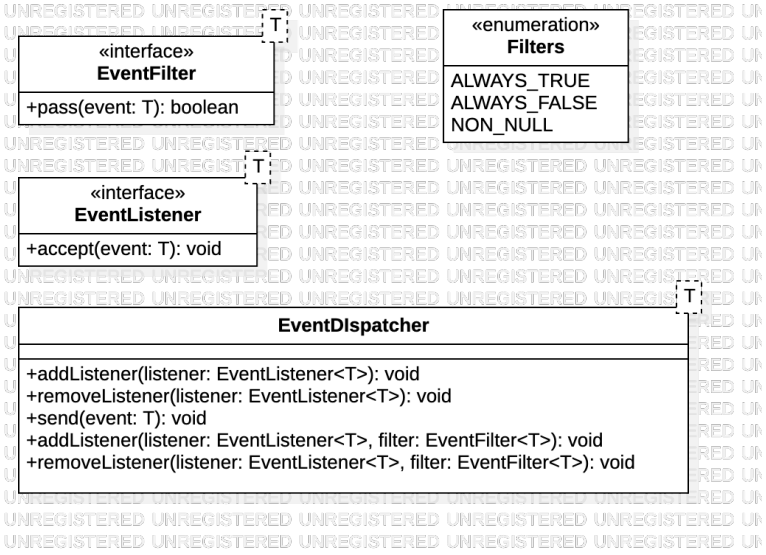

FIGURE 2 – UML

Implémentez le tout en utilisant une énumération de filtres par défaut : celui qui renvoie toujours vrai, toujours faux ou qui teste si l'évènement n'est pas `null`.

Question n°4

Désormais on se propose de chaîner les dispatchers de différents types de sorte qu'un évènement de type `T` produit sur un dispatcher de type `T` qui serait chaîné à un dispatcher de type <S> transmette l'évènement `T` sous la forme `S` au dispatcher de type `S` (qui lui-même...). Cela sous-entend nécessairement qu'au chaînage est associé un objet de type `Converter<T,S>`, comme l'indique la méthode générique :
`<S> void addNextEventDispatcher(EventDispatcher<S> d, Converter<T,S> f);`.

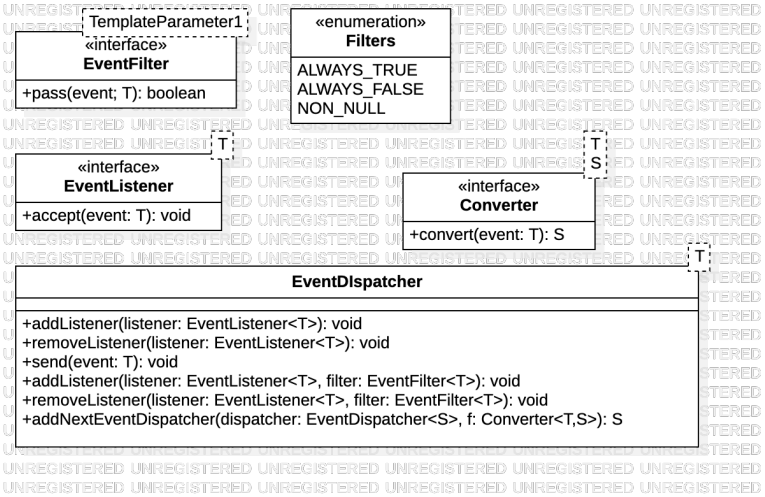

FIGURE 3 – UML

Implémentez et testez.

Note : Le main de test pourrait ressembler à :

```
EventDispatcher<Integer> ai = new EventDispatcher<>(); // Integer dispatcher
EventDispatcher<String>  as = new EventDispatcher<>(); // String dispatcher
Converter c = new Converter<Integer,String> {
    public String convert(Integer i) {
        return "string;␣"+i;
    }
ai.addNextEventDispatcher(as, c); // chain i-dispatcher to s-dispatcher
// add some listeners to both ai and as
ai.send(666); // send i-event to i-listeners, convert, and send to s-listeners...
```

Note : Prendre soin de bien utiliser autant que possible les contraintes de généricité pour une utilisation la plus large possible de ces types.