M2 — LPM/PGF— TP n°1
Java générique

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

9 octobre 2025

Comment procéder ?

Il est obligatoire d'effectuer le travail en équipe (idéalement 3 personnes : un prompteur, un développeur, un testeur).

Il est obligatoire d'utiliser `git` (le gitlab de l'UFR) et d'inviter l'enseignant dans le projet git correspondant, le dépôt devra être soigné (documentation en markdown, etc).

Il est obligatoire de réaliser des tests unitaires et fonctionnels ; décrivez-les clairement. Les tests devraient être réalisés à l'aide de JUnit.

Le code doit être commenté en Javadoc, etc.

Il est très fortement encouragé d'utiliser une IA, mais attention :

- toute utilisation d'une IA doit être mentionné dans le résultat (commentaire du code, fichiers accompagnants le code),
- l'historique de toutes les interactions avec l'IA doit être conservé (aka le prompt),
- l'IA ne doit pas servir à solutionner intégralement le problème. Donc, ne pas couper/coller tout ou partie du sujet, imaginez que la spécification du problème est de l'ordre du secret industriel et ne doit donc pas sortir à l'extérieur.
- l'IA ne doit être qu'un appui utile ; décrivez donc finement ce dont vous avez besoin avec vos propres mots, affinez les réponses. Il faut essayer de coder par soi-même et utiliser l'IA pour analyser la solution, l'affiner, débrouiller un problème technique, etc.
- toute réponse de l'IA doit être questionnée et vérifiée.

Chaque exercice doit être contenu dans un répertoire différent avec des sources autonomes (tout l'exercice n°1 dans le répertoire src1, etc).

Bref, il faut essayer d'effectuer et rendre un travail de niveau «professionnel».

Rendu final

Le rendu sera effectué par un clonage du dépôt git à une date qui vous sera indiquée en temps et en heure. Le dépôt devra comprendre un fichier `BINOME.txt` contenant le nom et le prénom (dans cet ordre !) des élèves concernés.

Généralités

Dans ce TP, on vise à construire un système de distribution de messages.

Attention, ce TP ne doit pas utiliser les λ-expressions.

Exercice n°1 - Généricité simple

Pour un système donné :

- les messages sont d'un type désigné par `T` ;
- afin de recevoir un évènement, un objet doit être du type `Receiver<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 autre traitement de la donnée).
- la distribution s'effectue *via* des instances de la classe `Sender<T>`. Afin qu'un évènement puisse être reçu par un `Receiver`, il faut au préalable que celui-ci **s'enregistre** auprès du `Sender` *via* sa méthode `add`. La **distribution** d'un évènement à tous les listeners enregistrés s'effectue par appel à la méthode `send` du `Sender` concerné. Enfin, il est possible à un récepteur de se désabonner par appel à la méthode `remove` du `Sender` auprès duquel il s'était préalablement enregistré. L'enregistrement et le désenregistrement recoivent un booléen indiquant si l'opération a réussi ou non.

Ce qui peut-être résumé par les deux diagrammes UML qui suivent.

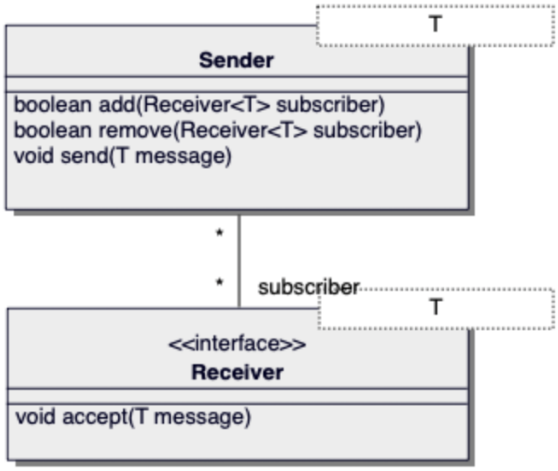

FIGURE 1 – Diagramme statique

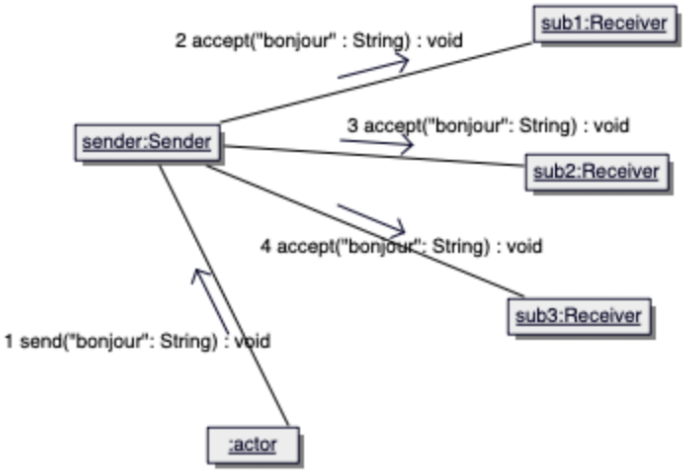

FIGURE 2 – Diagramme de collaboration

1. Implémentez l'interface et la classe décrite (complétez avec n'importe quel champ supplémentaire mais ne pas rajouter de méthode).
2. Dans une classe `Test` et son `main`, testez un assemblage permettant d'envoyer des messages de type `String` :
   — distribuez un évènement à un `Sender` vide,
   — ajoutez un récepteur et envoyez un évènement,
   — ajoutez un autre récepteur, envoyez un évènement,

— supprimez le premier récepteur, envoyez un évènement,
— etc.

3. Testez un assemblage avec des messages de type `Integer`. Même scénario de test que précédemment.
4. Créez un type `record` nommé `Journal` contenant un composant `article` de type `String`. Testez la distribution toujours selon le même scénario mais avec des messages de type `Journal`
5. Pouvez-vous abstraire les tests, pour fabriquer une méthode de test générique ? Si oui, implémentez-là.
6. Pouvez-vous créer une classe générique d'implémentation (simple) de `Receiver` ?

Exercice n°2 - Covariance et Contravariance

Dans cet exercice, seule la classe `Sender` est (éventuellement affectée).

1. Modifiez le TP précédent de sorte que l'on puisse faire tous les assemblages raisonnables. Pour vous guider, posez-vous des questions comme «puis-je utiliser un `Receiver<Integer>` avec un `Sender<Number>` ?» ou l'inverse ? Implémentez, testez.
2. Modifiez la classe `Sender` de sorte que l'on puisse faire un envoi groupé de messages *via* une méthode `sendAll(List<T> msgs)`. Implémentez, testez.
3. Le paramètre de `sendAll` est-il correctement typé ? Peut-on ouvrir pour augmenter le nombre de types acceptables ? Implémentez, testez.
4. Ajoutez dans `Sender` une méthode permettant de récupérer l'historique de tous les messages envoyés depuis le début de l'existence du `Sender` avec la méthode `getHistory(List<T> msgs, int n)`. Le paramètre n indique qu'on ne souhaite que les (au plus) n derniers messages.
5. Le paramètre de `getHistory` est-il correctement typé ? Peut-on ouvrir plus grandement ? Implémentez, testez.

Exercice n°3 - Héritage et généricité

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

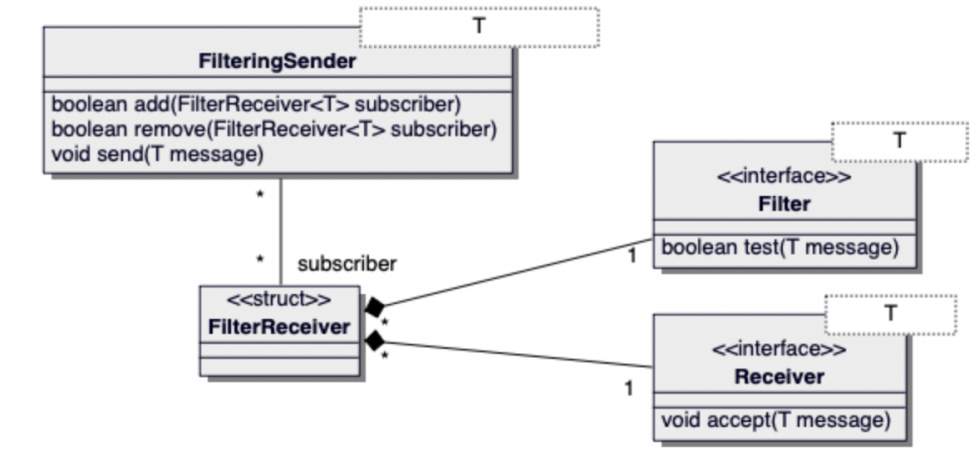

FIGURE 3 – UML

1. Implémentez le `record FilterReceiver`,
2. Implémentez la classe `FilteringSender`,
3. Créez une énumération de filtres par défaut, `ALWAYS_TRUE`, `ALWAYS_FALSE`, `NOT_NULL`, `STRING_EMPTY` et testez-les.
4. Est-il possible de construire la classe générique `FilteringSender<T>` à partir de la classe `Sender<T>` (sans la modifier) ? L'implémenter par exemple sous le nom `FilteringSenderByInheritance<T>`.

Exercice n°5 - Chaînage

Désormais on se propose de chaîner des senders de différents types de sorte qu'un évènement de type `T` produit sur un sender de type `T` à qui serait chaîné à un `Sender` de type `S` lui transmettrait l'évènement converti. Cela sous-entend nécessairement qu'au chaînage est associé un convertisseur de type `Converter<T,S>`, comme l'indique la méthode générique :
`<S> void add(Sender<S> d, Converter<T,S> f);`.

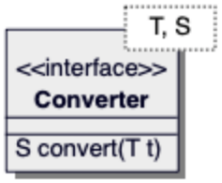

FIGURE 4 – UML

Implémentez et testez.

Note : Le main de test pourrait ressembler à :

```
var si = new Sender<Integer>(); // Integer sender
var  ss = new Sender<String>(); // String sender
var c = new Converter<Integer,String> {
    public String convert(Integer i) {
        return "string; "+i;
    }
ai.add(ss, c); // chain i-sender to s-sender
// add some listeners to both ai and as
ai.send(666); // send, convert...
```

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

Exercice n°6 - Extensions

Proposer des extensions du système.