Event bus Article

Introduction

L'un des mantras du développement est le découplage qui consiste à réduire les dépendances entre composants.

Pourquoi un fort couplage est problématique ?
Si un composant est dépendant à de nombreux autres, cela va rendre plus difficile l'écriture de tests unitaires car l'isolation du composant testé sera plus difficile à mettre en œuvre.

Exemple:

```
public class MyService {
    private final ServiceA serviceA;
    private final ServiceB serviceB;

    public MyService(ServiceA serviceA, ServiceB
serviceB) {
        this.serviceA = serviceA;
        this.serviceB = serviceB;
    }

    // Suite du code
}
```

Une possibilité est de « mocker » les composants mais cela implique, souvent, de rendre dépendant le test unitaire de l'implémentation du composant testé. Cette forte dépendance à l'implémentation rend le test unitaire fragile.

Exemple:

```
public class MyService {
    private final ServiceA serviceA;
    private final ServiceB serviceB;

    // début du code
    // ...

    public void aMethod(...) {
        // ...

        // La méthode ci-dessous doit être mockée pour
tester le service
        // Mocker une telle méthode entraine un couplage
entre l'implémentation du service et ses tests
        serviceB.callComplexMethod(...);
    }
}
```

La conséquence directe d'avoir des tests fragiles est leur difficulté de maintenance qui entraîne une baisse de motivation de leur écriture. La situation se résumera à : Moins de tests, plus d'incertitude, plus de régression, plus de maintenance.

En dehors des tests, avoir de nombreuses dépendances entre composants augmente l'adhérence de leur cycle de vie et nous avons des changements d'un composant impactant le code des autres composants. Cela peut être limité en respectant les principes SOLID, principalement ceux liés à l'utilisation d'interfaces. Mais limitation d'un problème ne signifie pas suppression de celui-ci.

```
public interface ServiceB {

    // La méthode ci-dessous a été modifiée pour
améliorer la qualité du code
    // void doOneThing(String email);

    // Le problème est que les composants utilisant ce
service vont être impactés.
    // Ce genre de situation peut entrainer la peur de
modifier l'interface.
    void doOneThing(Email email);
}
```

Voyons maintenant un exemple de code métier avec des dépendances et étudions comment nous pouvons les réduire.

Use case

Contexte

Une entreprise B2B propose un service de livraison de données.
Cette livraison prend la forme d'un fichier (csv ou Excel) mis à disposition du client et contient les données d'une population. Le terme population désigne un ensemble de critères permettant de sélectionner les données à livrer. Un exemple de population est « les PME dans la région Bretagne ayant moins de 50 salariés ».

Description du service

Une livraison suit les étapes suivantes:

- Réception de la commande du client
- Envoi du mail de prise en compte de la commande
- Vérification de la taille de la commande
  - Si la commande est importante envoi d'un mail de validation par le commerce.
  - Si la commande est refusée alors la commande est annulée et un

mail est envoyé au client
- Production du fichier à restituer soit au format csv ou Excel
- Envoi du mail de livraison

Implémentation du service

Le code ci-dessous est une version simplifiée.

Exemple:

```
public class OrderService {
    private final MailService mailService;   // Service
d'envoi de mail
    private final OrderProcessor processor;  // Traite
la commande

    public OrderService(MailService mailService,
OrderProcessor processor) {
        this.mailService = mailService;
        this.processor = processor;
    }

    // Traite une commande
    public void processOrder(Order order) {

        mailService.sendAwarenessEmail(order);

        OrderRules rules = new OrderRules();  // Gère
les règles d'acceptation des commandes

        if (!rules.accept(order)) {
            mailService.sendValidationEmail(order);
            return;
        }

        processor.deliver(order);
    }

    // Confirme le traitement d'une commande dont la
confirmation a été demandée
    public void confirmOrder(Order order) {
        processor.deliver(order);
    }

    // Refuse le traitement d'une commande
    public void refuseOrder(Order order) {
        mailService.sendRefusalEmail(order);
    }
```

```
}
```

Avec ce code, nous voyons que nous avons des dépendances aux composants d'envoi de mail, de traitement de commande et également une dépendance avec les logs métiers. Cette dernière dépendance apporte une certaine lourdeur dans la lecture du code.

Ces dépendances impliquent que lors des tests du service, nous aurons besoins d'instances des composants. Ces instances seront, dans pratique, mocked ou bien stubbed (désolé pour les anglicismes).

Changeons de vison

Le code précédent répond à une logique impérative: on réalise les actions de manière séquentielle.

Changeons de vision, en adoptant une logique d'échange de messages. Si on reprend le code du service avec cette logique, nous obtenons (en pseudo-code, les messages sont placés entre crochets):

Exemple:

```
public class OrderService {

    // Traite une commande
    public void processOrder(Order order) {

        [ Une commande est reçue ]

        OrderRules rules = new OrderRules();  // Gère
les règles d'acceptation des commandes

        if (!rules.accept(order)) {
            [ La commande est bloquée ]
            return;
        }

        [ Demande la réalisation de la commande ]
    }

    // Confirme le traitement d'une commande dont la
confirmation a été demandée
    public void confirmOrder(Order order) {
        [ Demande la réalisation de la commande ]
    }

    // Refuse le traitement d'une commande
    public void refuseOrder(Order order) {
        [ La commande est refusée ]
```

```
    }
}
```

Avec ce code, nous constatons que nous avons réduit le couplage entre le service et les autres composants.
Nous constatons également que nous avons 2 types de messages: les événements et les commandes.
La différence entre ces types est la suivante:

- un événement **s'est produit**
- une commande est une action **à réaliser**

La question est comment implémenter ce pseudo code.

La pattern publish / subscribe

Une description de ce pattern se trouve sur la page suivante: https://fr.wikipedia.org/wiki/Publish-subscribe

Le point de ce pattern est de découpler le publisher de ses subscribers. Le publisher envoie un message sans se soucier de « receveurs » de celui-ci.
L'extension de ce pattern est l'event bus.

Event bus

Un event bus est un composant utilisé comme médiateur entre différents composants.
Chaque composant peut envoyer des messages à l'event bus qui se charge de les transmettre aux composants s'étant abonnés.

Il existe quelques implémentations d'event bus en Java: celle proposée par Google via sa librairie Guava, Mycila, GreenRobot (orienté Android) et Mbassador. Ce dernier est celui sélectionné dans la suite de cet article https://github.com/bennidi/mbassador

> *Pourquoi avoir choisi Mbassador ?*
> *Cette librairie permet de gérer les appels synchrones et asynchrones, permet de filtrer les messages, supporte les weak references. Tout cela de manière simple et élégante (avis personnel).*

Prenons le bus

Avant de rentrer dans les détails, voici le code du service que nous obtenons:

```
public class OrderService {

    private final MBassador<Message> bus;

    public OrderService(MBassador<Message> bus) {
        this.bus = bus;
```

```
    }


    public void processOrder(Order order) {
        bus.publish(new OrderReceivedEvent(order));

        OrderRules rules = new OrderRules();

        if (!rules.accept(order)) {
            bus.publish(new OrderBlockedEvent(order));
            return;
        }

        bus.publish(new ProcessOrderCommand(order));
    }

    public void confirmOrder(Order order) {
        bus.publish(new ProcessOrderCommand(order));
    }

    public void refuseOrder(Order order) {
        bus.publish(new OrderRefusedEvent(order));
    }
}
```

Comme nous le voyons dans ce code, le service est maintenant complètement découplé à l'exception de l'event bus. Le service envoie vers le bus les différents types de messages: évènement et commande.

Les messages envoyés au bus sont interceptable par tous les composants prêts à les prendre en charge. Avec Mbassador, cette déclaration de prise en charge via les méthodes définies comme des « handlers ».

Exemple où chaque handler gère un type de message:

```
public class MailService {
    @Handler
    public void sendAwarenessEmail(OrderReceivedEvent
message) {
        System.out.println("Email informant le client de
la réception de la commande " + message.order());
    }

    @Handler
    public void sendValidationEmail(OrderBlockedEvent
message) {
        System.out.println("Email envoyé en interne pour
valider la commande " + message.order());
```

```
    }

    @Handler
    public void sendRefusalEmail(OrderRefusedEvent
message) {
        System.out.println("Email envoyé au client pour
l'informer du refus du traitement de la commande " +
message.order());
    }

    @Handler
    public void sendLinkToOrderFile(OrderDeliveredEvent
message) {
        System.out.println("Envoi du mail avec le lien
du fichier pour la commande " + message.order());
    }
}
```

Exemple d'un service de log de messages métiers qui gère de la même manière tous les messages:

```
public class DomainLogging {
    @Handler
    public void logDomainMessage(Message message) {
        System.out.println("Log domain message: " +
message);
    }
}
```

Un handler définit le *listener* qui reçoit les messages du bus. Il reste maintenant à informer le bus qu'elles sont les classes susceptibles d'être intéressées par les message.
Cette opération se fait en déclarant les abonnements de la manière suivante:

```
bus.subscribe(mailService);
bus.subscribe(domainLogging);
// suite des abonnements
```

Avantages d'utiliser un Event Bus

Le principal avantage est le découplage des composants qui simplifie les tests et l'ajout de nouveaux comportements.
Imaginons qu'une nouvelle demande concerne l'ajout de statistiques concernant les commandes (reçues, refusées, en attente d'approbation).
Dans notre contexte, cela consistera à définir un nouvelle classe avec les handlers nécessaires et d'abonner cette nouvelle classe au bus.
Cet ajout n'aura aucun impact sur le code existant.

Exemple de classe pour les statistiques:

```
public class StatsUpdate {
    @Handler
    public void
updateReceivedOrderStat(OrderReceivedEvent message) {
        System.out.println("Maj de la stats du nombre
total de commandes avec la commande " +
message.order());
    }

    @Handler
    public void updateRefusedOrderStat(OrderRefusedEvent
message) {
        System.out.println("Maj de la stats du nombre de
commandes refusées avec la commande " +
message.order());
    }

    @Handler
    public void updateBlockedOrderStat(OrderBlockedEvent
message) {
        System.out.println("Maj de la stats du nombre
total de commandes demandant une confirmation avec la
commande " + message.order());
    }
}
```

Abonnement de la classe:

```
bus.subscribe(mailService);
bus.subscribe(domainLogging);
bus.subscribe(statsUpdate);
// suite des abonnements
```

Inconvénients d'utiliser un Event Bus

Tout choix technique vient avec ses avantages mais, également avec ses inconvénients.

Le principal inconvénient est le changement de paradigme qui oblige à réfléchir en termes d'échanges de messages.
Cela implique à prendre du temps pour bien définir les messages.
De cet inconvénient, en découle un autre: lors de la lecture du code, on ne voit pas toutes les opérations réalisées: il faut faire le lien entre opération et message.