**Cas concret de l’importance du langage omniprésent (Ubiquitous Language) et de son influence sur le découpage des microservices**

**Contexte**

Dans une application e-commerce, différents termes sont utilisés pour représenter les mêmes concepts dans différentes équipes ou départements. Par exemple, le **service commercial** appelle une commande "Order", tandis que l'équipe logistique parle de "Shipment" pour désigner une commande prête à être expédiée. L'équipe marketing, quant à elle, utilise le terme "Purchase" pour parler des ventes. Ces différences terminologiques peuvent sembler triviales au premier abord, mais elles risquent de créer des incompréhensions profondes lorsque l'on découpe l'application en microservices.

**Problème : Mauvais découpage des microservices**

Sans un **langage omniprésent** et une **compréhension partagée** de la terminologie, chaque équipe développe ses propres services sans clarifier le sens exact des concepts qu'elle manipule. Cela conduit à un **mauvais découpage** des microservices, car les frontières des services ne sont pas définies en fonction des **responsabilités métiers**, mais en fonction des termes mal compris ou interprétés différemment.

**Cas concret : Mauvaise gestion des "Commandes"**

Dans cet exemple, la commande (order) est un concept central dans l'application e-commerce. Cependant, chaque équipe (commercial, logistique, marketing) utilise un terme différent pour parler de la commande. Le découpage suivant est alors décidé :

1. **Service Commercial (Order Service)** : Ce service est censé gérer les commandes, mais il ne traite que la partie **création** des commandes. On suppose qu'une commande se limite à la demande initiale du client, et donc le service ne gère pas le suivi ni l'expédition.
2. **Service Logistique (Shipment Service)** : Ce service gère l'expédition des commandes. Cependant, l'équipe logistique a modélisé cela en se basant uniquement sur les expéditions, en les traitant comme des objets indépendants des commandes. Ils n'ont pas bien compris que l’expédition est une étape dans la vie d’une commande et non un concept totalement séparé.
3. **Service Marketing (Purchase Service)** : L’équipe marketing crée un service pour gérer les "purchases" (achats), qui, selon leur perspective, inclut non seulement les commandes des clients, mais aussi toutes les campagnes marketing et promotions associées. Ils modélisent cela comme un service distinct, alors qu’il aurait pu être intégré dans un service plus large lié aux commandes.

**Conséquences du mauvais découpage**

1. **Données dupliquées** : Chaque service finit par avoir sa propre représentation des "commandes", entraînant la **duplication des données** et une divergence des informations. Par exemple, le service commercial traite les commandes sans avoir connaissance de l'état de leur expédition, et le service logistique crée ses propres "expéditions" sans se synchroniser correctement avec les données des commandes commerciales.
2. **Couplage fort et intégrations complexes** : Comme les services ne sont pas bien découpés et ne partagent pas une compréhension commune de ce qu'est une "commande", ils doivent constamment s'appeler pour obtenir des informations manquantes. Cela entraîne un **couplage fort** entre les services, nécessitant des intégrations complexes et des appels synchrones pour mettre à jour ou consulter l'état des commandes.
3. **Incohérence des données** : Chaque service gérant une partie différente du cycle de vie de la commande sans coordination claire, cela provoque des **incohérences**. Par exemple, une commande pourrait être marquée comme "expédiée" par le service logistique, alors que pour le service commercial, elle est toujours en attente de validation du paiement.
4. **Problèmes de scalabilité** : Ce mauvais découpage rend difficile la scalabilité. Par exemple, si le nombre de commandes augmente, le **service logistique** et le **service commercial** doivent faire face à des charges accrues en raison de leur **mauvaise séparation des responsabilités**. Il devient difficile de faire évoluer un service sans impacter les autres.

**Solution avec un langage omniprésent**

Si un **langage omniprésent** avait été adopté dès le départ, les équipes auraient clarifié le concept de "commande" et défini ensemble une terminologie partagée pour le domaine. Cela aurait conduit à un découpage plus logique et plus cohérent des services.

1. **Clarification des termes** : Grâce au langage omniprésent, les équipes se seraient mises d'accord sur le fait qu'une **"commande" (order)** représente une entité unique avec plusieurs étapes dans son cycle de vie :
   * **Création de la commande** (par le client)
   * **Traitement du paiement**
   * **Expédition de la commande**
2. **Meilleur découpage des microservices** :
   * **Service de commande** : Ce service gère **tout le cycle de vie** d'une commande, depuis sa création jusqu'à son expédition, en intégrant le suivi du paiement et de la logistique. Il utilise des événements de domaine pour notifier les autres services (comme le service marketing ou le service logistique) des changements d'état des commandes.
   * **Service logistique** : Plutôt que de gérer des entités "expédition" indépendantes, ce service écoute les événements "Commande prête à être expédiée" émis par le **service de commande** et s'occupe de l'expédition proprement dite, en envoyant des mises à jour au service de commande.
   * **Service marketing** : Ce service n’a plus besoin de gérer les "achats" de manière isolée. Il écoute les événements comme "Commande confirmée" ou "Commande expédiée" pour déclencher des campagnes ou des promotions.

**Résultats d'un bon découpage avec DDD et langage omniprésent**

1. **Données cohérentes** : En ayant une définition claire et partagée de la commande, chaque microservice peut interagir avec le concept de manière cohérente. Les services peuvent réagir aux événements sans dupliquer les données ou créer de divergences.
2. **Faible couplage** : Les microservices sont faiblement couplés, car chaque service se concentre sur son propre domaine et utilise des événements asynchrones pour réagir aux changements. Par exemple, le service de logistique ne connaît pas les détails internes de la commande, mais réagit uniquement lorsqu'une commande est prête à être expédiée.
3. **Meilleure scalabilité** : Chaque microservice est plus facile à faire évoluer indépendamment, car ils sont découplés de manière appropriée et ne partagent pas des concepts mal compris ou redondants.
4. **Simplicité d'intégration** : Les intégrations deviennent plus simples car elles sont basées sur des événements métier (comme "Commande confirmée", "Commande expédiée") partagés dans le langage omniprésent, plutôt que sur des appels synchrones complexes.

**Conclusion**

L'absence d'un **langage omniprésent** peut entraîner un mauvais découpage des microservices en créant des malentendus entre les équipes sur les concepts clés du domaine métier, ce qui conduit à des services mal conçus, fortement couplés et inefficaces. En adoptant un **langage partagé** dès le départ, aligné sur les termes et concepts métiers, les équipes peuvent découper les microservices de manière plus cohérente et faciliter la communication, la maintenabilité et la scalabilité du système.