Comment concevoir en orienté objet

Le RDD signifie "Responsability Driven Developement" et c'est un systeme qui permet de concevoir des applications en orienté objet. C'est une méthodologie assez abstraite qui se focus sur les "responsabilités" de chaque classes (appellées ici "candidats")

Pour mieux comprendre la suite il est donc important de se familiariser avec les concepts de candidats et responsabilités

- Une **responsabilité** est une obligation de faire une tâche ou de connaître quelque chose.
- Un **candidat** est une classe (ou tout composant logiciel) suceptible d'avoir des responsabilités

Identifier les fonctionalités

Use Cases

Image not found or type unknown

Tout d'abord il faut identifier les différentes fonctionalités du système, c'est à dire, du point de vue de l'utilisateur·ice. Qu'est ce que l'on peut faire avec l'application.

Dans l'exemple du laboratoire 6 de programmation orienté objet avec une caisse, lors de l'itération 1 on a les fonctionalités suivantes :

- Scanner un article sur base de la première lettre de son nom (SKU)
- Demander le total des articles scannés en comptabilisant leurs réductions de groupes

Cela ressemble donc un peu au processus d'identification des "Use Cases" en analyse.

Ensuite on peut répèter les étapes suivantes pour chaque fonctionalités du système.

Identifier les candidats

Stéréotypes des roles des candidats

Image not found or type unknown

Pour ce faire on peut commencer par identifier les différents stéréotypes des rôles des candidats et les garder dans un coin de sa tête :

1. Le détenteur d’informations connaît et fournit des informations. (va connaître, savoir des informations)
2. Le structureur maintient des relations entre les objets, éventuellement de même type, et des informations à propos de leurs associations.
3. Le fournisseur de services effectue un travail pour le compte d’un utilisateur. (va lire, afficher, calculer, fournir, ajouter, vérifier, etc)
4. Le contrôleur prend des décisions et pilote les actions de ses collaborateurs. (va contrôler, décider, coordonner, notifier, signaler, etc)
5. Le coordinateur réagit à des événements et délègue leurs traitements à d’autres. (va associer, regrouper, lister des éléments)
6. L’adaptateur convertit les données échangées entre l’application et un acteur externe. (par exemple un superviseur par rapport à une vue)

C'est intéressant de garder ces différents stéréotypes de rôles dans la tête pour identifier plus facilement les candidats. Dans notre projet, au vu de la description on peut **par exemple** (car il peut y avoir pleins d'interpretations différentes) identifier les candidats suivants :

- Une vue (pour l'interface)
- Un superviseur (**tip!** si il y a une vue il y a toujours un superviseur)
- Un panier de produits scannés
- Un catalogue des produits disponibles
- Un produit
- Une règle de prix (qui définis les réductions potentielles)
- Un SKU
- Un prix

A savoir que c'est possible d'en avoir moins, ici pour cette première itérations, les éléments produits, prix et SKU sont très peu voir absolument pas utile. Mais les prévoir quand même peut rendre le projet plus modulaire pour y ajouter de nouvelles fonctoinalités par la suite. Ce qui est l'une des raisons pour laquelle il est important de savoir clairement toutes les fonctionalités à représenter.

Identifier les responsabilités

Classification des responsabilités

Une fois que l'on a les candidats potentiels on peut maintenant essayer de trouver les responsabilités qui vont y être assignées.

Une responsabilité est représentée par un groupe **verbal**, donc on peut identifier des responsabilités potentielles avec les mots (et catégories) suivantes par exemples :

- Une connaissance (connaître, savoir)
- Une connexion entre objets (associer, regrouper, lister)
- Un service (lire, afficher, calculer, fournir, ajouter, vérifier, parcourir, convertir, etc)
- Une prise de décisio (contrôler, décider, coordonner, notifier, signaler)

Dans le cas de ce projet on a identifier les responsabilités suivantes :

- Régis au scan d'un produit
- Demander le montant total
- Afficher le montant total
- Ajouter un produit au panier
- Récupérer le montant total
- Récupérer un produit dans la catalogue
- Calculer le prix total
- Regrouper la liste des produits scannés
- Regrouper la liste des produits disponibles
- Fournit un produit sur base de sa première lettre
- Connais une règle de prix
- Connais un SKU
- Connais le nom du SKU
- Connais la première lettre du SKU
- Calcule le prix pour un certain nombre d'articles (produits)
- Connais son prix unitaire
- Connais sa règle de "combo de prix" (réductions)
- Connais sa valeur
- Connais sa devise

Verifier les candidats et les responsabilités

Ensuite on peut relire les responsabilités et les candidats pour s'assurer qu'il n'y a pas de doublons.

Ensuite on peut essayer d'associer les différentes responsabilités aux différents candidats.

Enfin on peut mettre des stéréotypes de rôles sur les candidats (voir plus haut). Et il faut de préférence que chaque candidat aie 2 stéréotypes (il peut y en avoir 3 mais certainement pas moins de 2 ou plus de 3)

Si un candidat n'est pas valide il faut donc essyer de voir qui peut se charger des responsabilités, fusioner avec un autre candidat. Ou dans le cas contraire où il y aurait trop de responsabilités pour un candidat, de les diviser dans différents candidats ou en créer des nouveaux.

Créer des cartes CRC et les lier entre elles

Exemples de cartes CRC

Une fois que l'on a notre liste de candidats et responsabilités vérifiées, on peut les associer pour de bon sur des cartes **CRC** (Classe Responsabilité Collaborateur). Chaque carte va représenter l'un des candidat et va contenir :

1. Le nom
2. La liste des responsabilités associées
3. La liste des autres classes avec qui elle va intéragir pour respecter ses responsabilités

Ainsi on peut d'abord lier les candidats et les responsabilités puis ensuite lier les cartes CRC entre elles. Pour ce faire on peut essayer de simuler le chemin de l'action d'un·e utilisateur·ice. Imaginons pour scanner un article par exemple :

1. On scanne un produit (Vue → Superviseur)
2. On récupère l'item dans le catalogue (Superviseur → Catalogue)
3. On ajoute l'item au panier (Superviseur → Panier)

Et on fait de même pour le calcul du total :

1. On demande pour avoir le total (Vue → Superviseur)
2. On récupère le montant total (Superviseur → Panier)
3. On calcule le prix pour X nombre d'un même produit (Panier → Produit → Règle de prix → Prix)

Ainsi on obtien la structure suivante :

Cartes CRC

Création d'un schéma de collaboration

On peut ensuite représenter les différentes classes de façon à faire des liens entre elles. Et on peut ensuite décrire les appels de fonctions qui vont être fait entre elles pour chaque fonctionalité.

Par exemple pour scanner un article :

Chemin	Appel de fonction
Utilisateur → Vue	Scan d'un article
Vue → Superviseur	`onScan(n)`
Superviseur → Catalogue	`getProduct(n)`
Superviseur → Panier	`addProduct(Product)`

Et pour obtenir le total :

Chemin	Appel de fonction
Utilisateur → Vue	Demande le total
Vue → Superviseur	`onCheckout()`
Superviseur → Panier	`getTotal()`
Panier → Produit	`getPrice(5)`
Produit → Regle de prix	`getPrice(5)`
Superviseur → Prix	`getDevise()`
Superviseur → Vue	`setTotal(50, "EUR")`

On peut représenter tout ceci avec le diagramme suivant :

Diagramme de collaboration

Ainsi on doit pouvoir couvrir l'intégralité des responsabilités, des fonctionalités et des candidats/classes.

On peut aussi vérifier ici que la règle de Demeter est bien respectée. La règle de Demeter consiste en :

- Chaque classe ne doit connaître des choses que sur les classes les plus "proches"
- Chaque classe ne parle qu'a ses attributs ou ce ses méthodes prennent en paramètre mais pas sur des retours de méthodes
- En somme chaque classe ne parle qu'aux objets de la même classe, des objets en attributs, des objets créés par cette dernière et les objets qu'elle a reçu en paramètre

La loi de Demeter est à nuancer toute fois, c'est à éviter (surtout quand c'est des communicatoins avec des classes très lointaines l'une de l'autre) mais des infractions à la loi de Demeter peuvent toute fois arriver. Par exemple ici la liaison entre Superviseur et Prix enfreint la loi de Demeter

Créer le diagramme de classe

Sur base de toutes les méthodes et classes définies, on peut maintenant les grouper dans des classes et y préciser les attributs dont elle a besoin (les informations qu'elle doit mémoriser)

Ensuite on peut faire les liens entre elles. C'est à dire faire les liens comme ceux précisés dans les cartes CRC, en y ajoutant les attributs.

Ainsi on peut vérifier qu'il n'y a pas de dépendence circulaire. Si il y en a ont peut essayer de faire une *inversion des dépendences* à l'aide d'une interface.

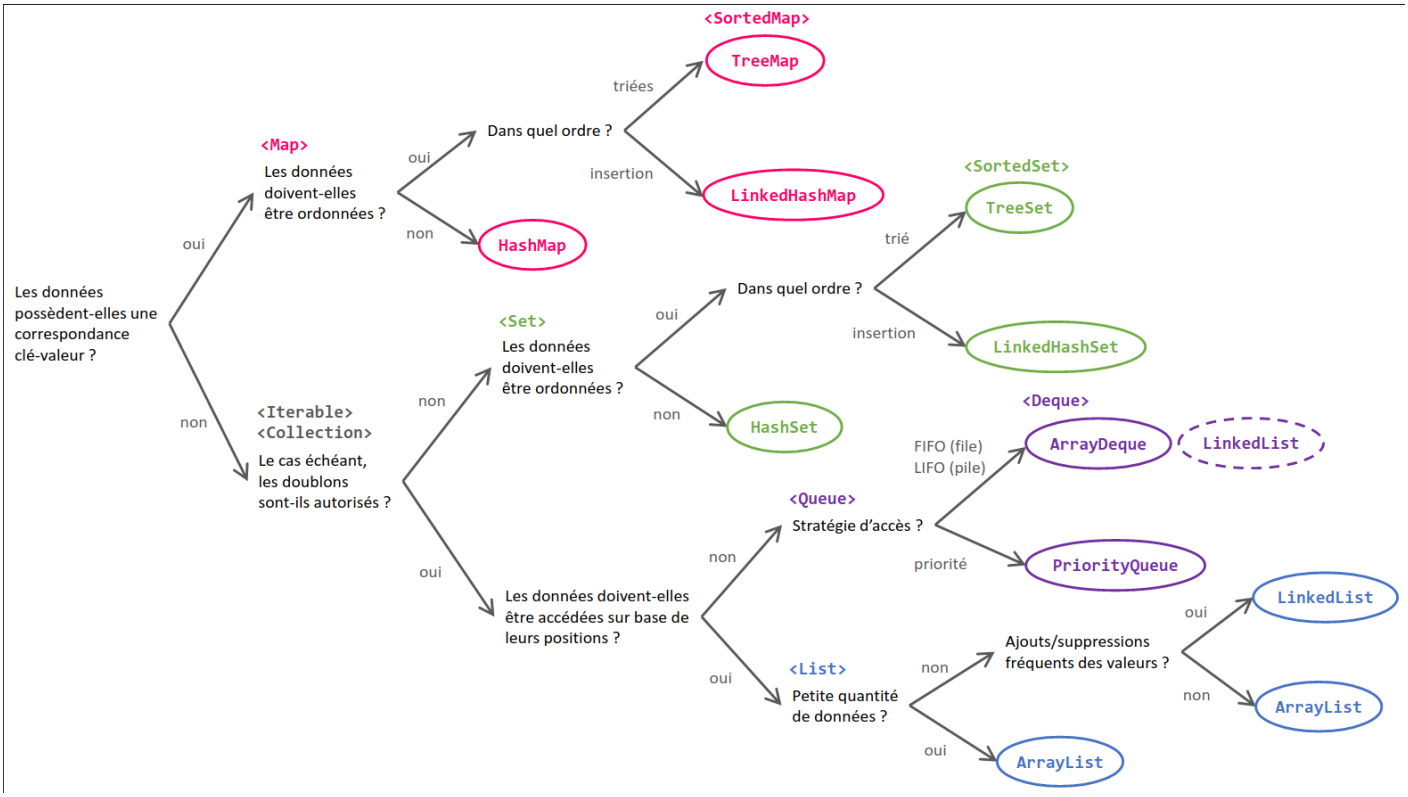
Par exemple ici "La vue compose le superviseur et le superviseur compose la vue" devient "Le Superviseur compose la vue, la vue implémente VueInterface et VueInterface compose Superviseur" ainsi il n'y a plus de dépendences circulaires.

Notre schéma final nous donne donc ceci :

Diagramme de classes

> La flèche en pointillé signifie "A implémente B" Les autres flèches pleines signifient "A compose B"

Donner des types et choisir les bonnes collections



Une fois que l'on connait les attributs, les méthodes, les classes et les responsabilités. On peut commencer à mettre des types sur les différentes donneés. Et quelque chose d'important en est de choisir la bonne collection pour représenter des collections d'éléments. L'image ci dessus devrait pouvoir vous aider à choisir la bonne.

Revision #5
Created 26 April 2023 12:31:36 by SnowCode
Updated 15 May 2023 06:33:10 by SnowCode