Natatoria
TP3 SE - M1 IL

Pierre Le Luron - Alan Turpin

30 novembre 2017

1 Postulats de départ

Des postulats sont donnés pour la conception de la piscine :

— Chaque composant de la piscine peut être représenté comme un service. Les clients attendent qu'un service soit disponible, l'utilisent puis libèrent la place.
— Les clients ne changent jamais d'avis : Ils entrent dans la piscine pour aller au bassin, restent un certain temps puis sortent de la piscine. Ils ne vont jamais se lasser et partir d'une file d'attente, ou sortir de la piscine sans être allés au bassin.
— On estime que la piscine est assez grande pour que la taille des files d'attente ne soit pas un problème.
— On compte sur les clients pour organiser les files d'attente eux-mêmes. Lorsqu'un client a fini d'utiliser un service, il notifie le ou les clients qu'il veut.
— Le temps de déplacement des clients d'un service à un autre n'est pas représenté. On estime que lorsqu'un client quitte un service, c'est comme s'il attendait déjà au prochain.
— On estime que les clients sont civils et n'essaient pas de voler du matériel ou de rester trop longtemps dans la piscine.

2 Conception

Nous avons choisi de représenter la piscine comme un conteneur de services, où chacun d'eux a sa classe spécifique. Chacune de ces classes contient une méthode d'entrée dans le service (où le client va attendre qu'il soit libre), une méthode d'utilisation du service et une méthode de sortie (où le client va libérer le service). Généralement, les méthodes d'entrée contiennent des appels à `wait()` pour attendre que le service soit disponible, et les méthodes de sortie contiennent un appel à `notify()` ou `notifyAll()` pour signaler à un ou plusieurs clients que le service est disponible. Ces méthodes sont synchronisées car c'est là qu'on lit ou modifie des variables partagées.

Le client est représenté comme un thread qui contient une référence vers la piscine pour pouvoir interagir avec ses services. Pour chaque service qu'il doit utiliser, il appelera donc les méthodes d'entrée, d'utilisation et de sortie dans cet ordre strict. Le thread se termine quand le client sort de la piscine.

2.1 Partie 1

La piscine contient 3 services : Le point de vente, les vestiaires et le bassin.

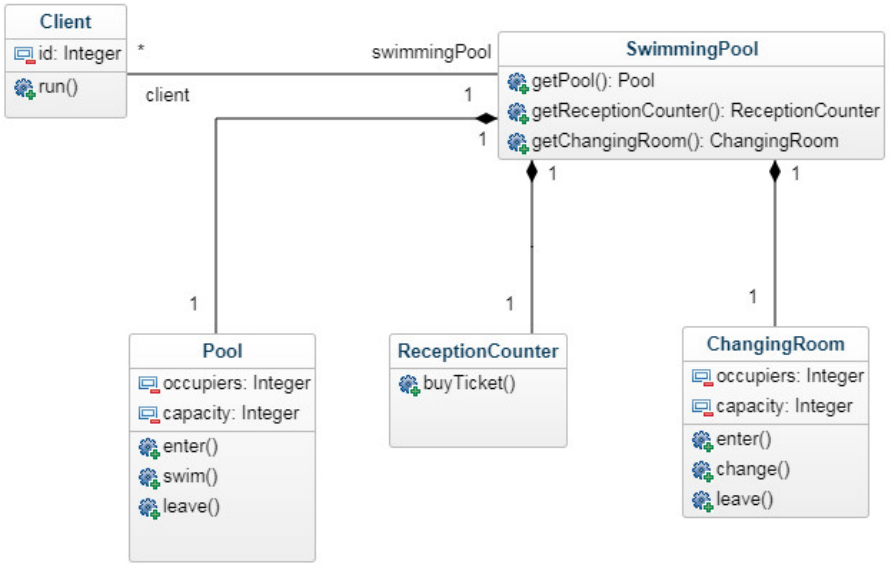

FIGURE 1 – Diagramme de classe de la partie 1.

Le point de vente n'autorise qu'un seul client à la fois : on peut alors appeler une unique méthode synchronisée `buyTicket()` sans `wait()` ni `notify()`.

C'est différent pour le bassin et les vestiaires : plusieurs clients peuvent utiliser ces services en même temps, mais il y a une certaine capacité à ne pas dépasser. Il faut donc un compteur pour les clients qui utilisent le service, et les fonctions d'entrée devront attendre si la capacité est atteinte. Les fonctions de sortie doivent à leur tour notifier un client qui attend car si quelqu'un libère un de ces services, c'est qu'il devient forcément disponible.

2.2 Partie 2

On change le point de vente pour qu'il y ait deux comptoirs, mais toujours une seule file d'attente.

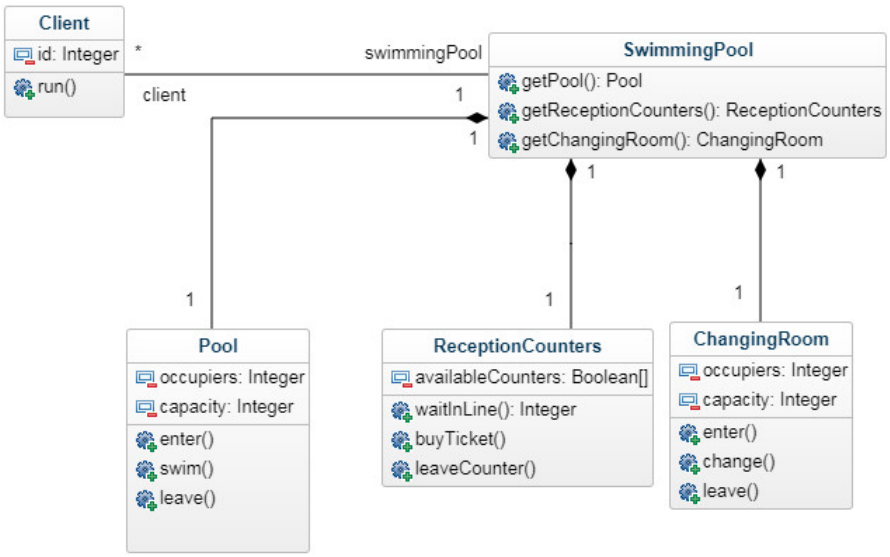

FIGURE 2 – Diagramme de classe de la partie 2.

Il faut donc ne pas stocker un compteur comme pour le bassin ou les vestiaires, mais plutôt des booléens qui indiquent la disponibilité des comptoirs. Les clients devront attendre si tous ces booléens sont vrais. Lorsqu'un client sera notifié, il cherchera parmi ces booléens lequel est faux,

puis le mettera à vrai. Une fois le ticket acheté, le booléen est remis à faux. Cela est implémenté avec une liste de booléens (pour autoriser éventuellement un nombre quelconque de comptoirs).

2.3 Partie 3

On ajoute un service de prêt de palmes. Les palmes doivent être empruntées et rendues par deux. Les clients peuvent choisir d'aller dans le bassin avec ou sans palmes.

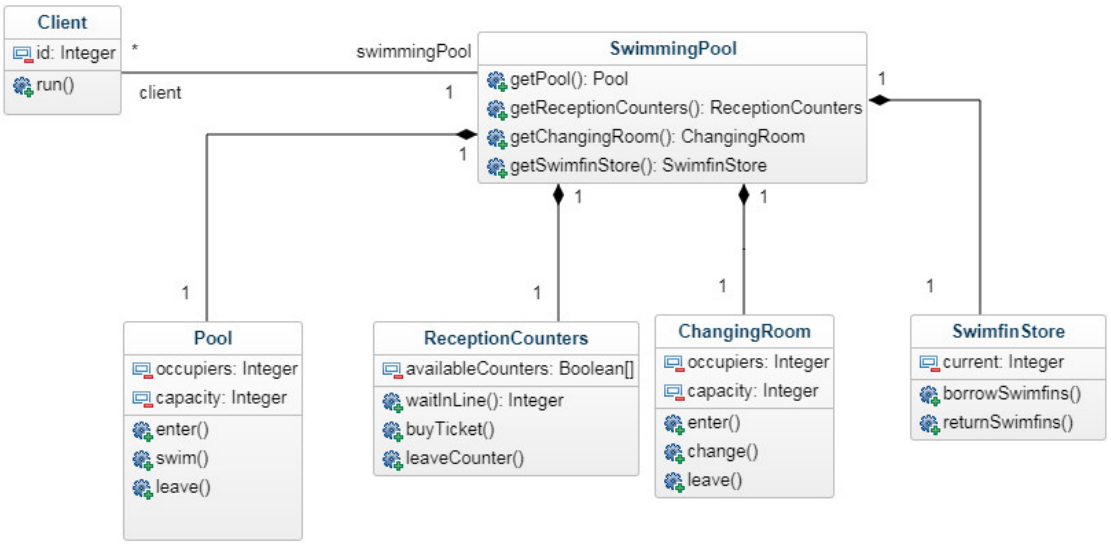

FIGURE 3 – Diagramme de classe de la partie 3.

Le stand de prêt de palmes fonctionne comme un compteur mais inversé : il y a un nombre initial de palmes que chaque client soustrait quand il emprunte, puis ajoute lorsqu'il rend la ressource. Pour simplifier les choses, il n'y a pas de problème du philosophe pour donner les palmes : elles sont données par paires. Les clients doivent attendre qu'une paire soit disponible, la prennent en une seule fois et la rendent aussi en une seule fois. On enlèvera et ajoutera 2 au compteur de palmes (on peut aussi représenter un compteur de paire à la place, si on est sûr qu'on a un nombre pair de palmes en réserve).

Le client appelle `Random.nextBoolean()` pour savoir s'il veut aller au bassin avec des palmes ou non. Dans le cas où il veut des palmes, il les rendra forcément après être allé au bassin. Un client qui n'en veut pas n'essayera pas de rendre les palmes qu'il n'a pas.

2.4 Bonus

Jean-Luc doit nettoyer le bassin de temps en temps. Il doit le faire quand le bassin est vide. Personne ne peut entrer dans le bassin quand Jean-Luc est en train de le nettoyer.

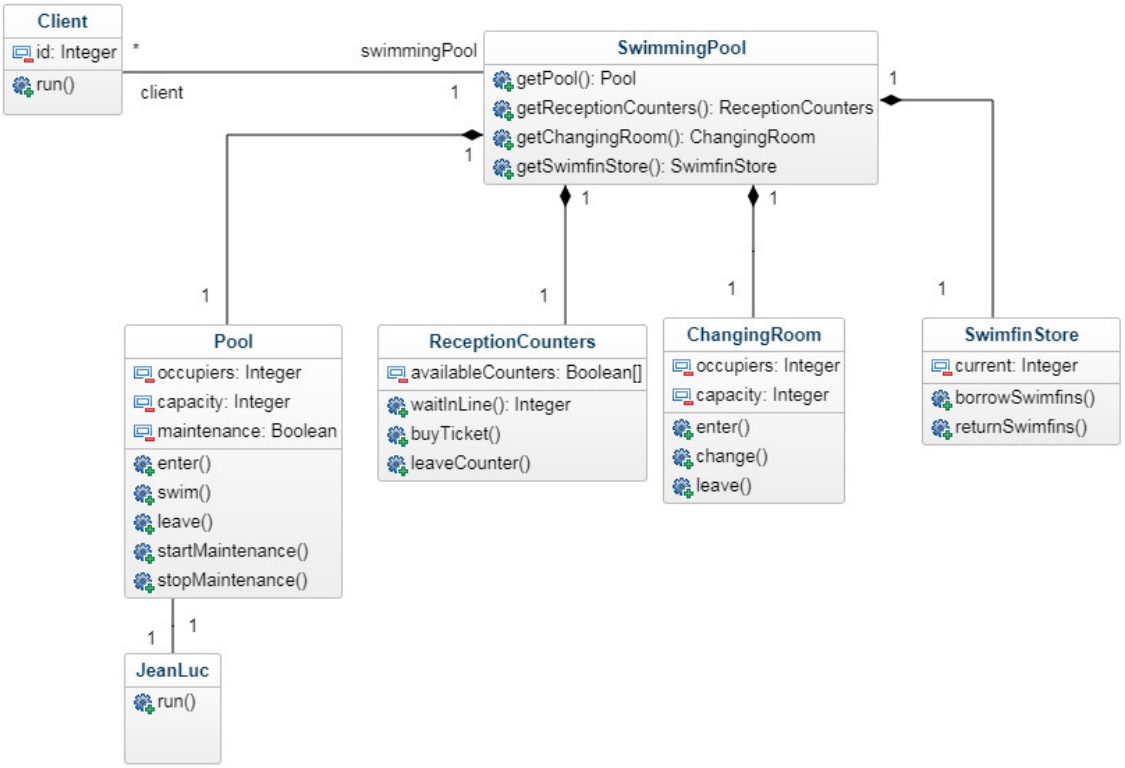

FIGURE 4 – Diagramme de classe du bonus.

Pour s'assurer que le bassin est vide, on veut que lorsque Jean-Luc veut nettoyer le bassin, plus personne ne soit autorisé à y entrer. Comme le client est roi, on ne veut pas non plus que Jean-Luc les sorte de force, alors il va simplement attendre qu'ils le fassent d'eux même (car on sait qu'ils sortiront un jour). Une fois qu'il n'y a plus personne, Jean-Luc peut se mettre à l'ouvrage, puis lorsqu'il a terminé, ré-autoriser les clients à entrer dans le bassin, jusqu'à la prochaine période de nettoyage.

Un "mode maintenance" est ajouté dans la classe du bassin pour interdire aux clients d'y rentrer. La méthode `startMaintenance()` permet d'activer ce mode et attendre que tout le monde soit sorti du bassin. La méthode `stopMaintenance()` désactive le mode maintenance et notifie tous les clients qui attendent avec `notifyAll()`, car ils peuvent maintenant entrer dans le bassin. Jean-Luc est implémenté comme un démon qui va tourner dans une boucle infinie et de temps en temps appeler ces deux méthodes. Le client devra aussi appeler `notifyAll()` quand il sort du bassin car il peut notifier un client qui veut entrer mais aussi Jean-Luc qui veut nettoyer le bassin.

3 Détails d'implémentation

Certains problèmes ont été rencontrés :

La JVM peut parfois réveiller des threads en attente sans qu'ils soient notifiés (*"spurious wakeup"*). Cela pose problème quand on veut faire un simple `if wait notify` et que la ressource est consommée par un thread pour laquelle la condition n'est pas satisfaite. Il faut donc corriger ça en re-vérifiant la condition avec un `while wait notify`.

Il est difficile de prédire quand une libération de ressource doit notifier un ou plusieurs threads. Dans le cas de la partie bonus, le fait d'avoir les clients qui notifient un seul thread lorsqu'ils sortent de la piscine peut fonctionner parfois, alors que ça ne doit pas être le cas. Le choix des temps de `sleep()` entre chaque action peut donner des hasards heureux où le programme "tombe en marche" plus souvent qu'il ne devrait et masque la potentielle erreur.

4 Traces

Les traces sont générées avec les paramètres suivants :
— Il y a 40 clients
— Acheter un ticket dure 2ms
— Les vestiaires ont une capacité de 4 personnes
— Mettre son maillot ou remettre ses vêtements dure 5ms
— Le bassin a une capacité de 10 personnes
— Nager dans le bassin dure entre 10ms et 20ms
— Le stand de prêts a un nombre initial de 16 palmes
— Emprunter et rendre une paire de palmes dure 2ms
— Jean-Luc attend 40ms avant de nettoyer la piscine
— Jean-Luc met 20ms pour nettoyer la piscine

Le nombre de clients est défini dans le fichier `Main.java`, les autres paramètres dans le fichier `SwimmingPool.java` dans chaque package.