Université du Québec à Chicoutimi
Département d'informatique et de mathématique
8INF957 – Programmation objet avancée : TP3

Rapport
Programmation Objet Avancée

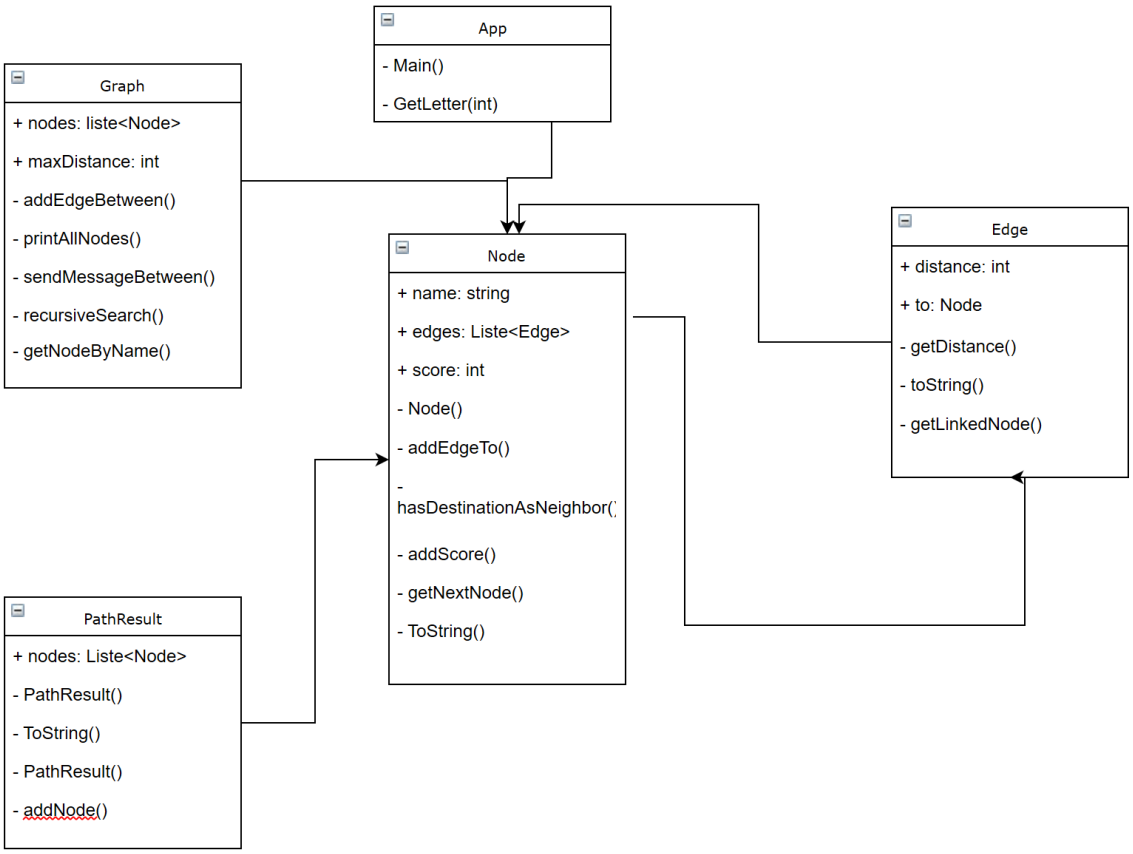
Choix 2

Georges COSSON, Augustin DE LA TAILLE & Antoine DEMON

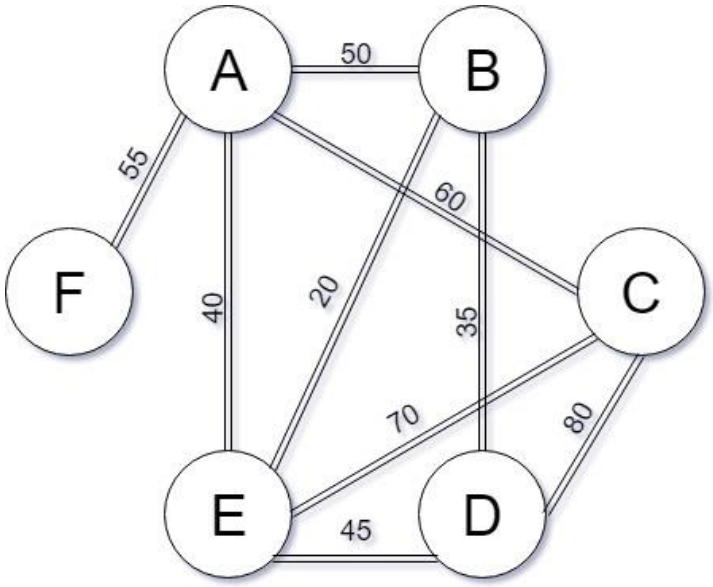
I. Question 1 :

1) Représentation :

Voici le diagramme UML des objets utilisés dans notre programme :



Nous avons créé un graph pour effectuer nos tests, puis nous l'avons changé pour valider notre code. Le graph actuel de vérification est celui-ci :



Dans nos tests actuels, nous envoyons des messages de F à D, puis de D à A.

Voici les résultats obtenus :

2) Algorithme

Nous avons mis en place un algorithme simple qui cherche de manière récursive un moyen pour le message de passer. Il se base sur le nœud actuel, cherche si la destination est atteignable (= lien direct + distance assez petite), si la destination n'est pas atteignable, alors il passe au nœud suivant (le nœud atteignable avec la plus petite distance). Il effectue la même logique qu'énoncée précédemment. Grâce à l'objet **PathResult**, nous gardons une trace de cette logique, que nous pouvons voir en console à la fin.

II. Question 2 :

Donnez un exemple de chaque concept de SOLID, vus en classe. Citez les avantages ce chacun de ces concepts.

Plusieur concepts de SOLID :

- **Single Responsibility Principle (SRP)**
- **Open Closed Principle (OCP)**
- **Liskov Substitution Principle (LSP)**
- **Interface Segregation Principle (ISP)**
- **Dependency Inversion Principle (DIP)**

1) Single Responsibility Principle

« Une classe doit avoir une seule raison de changer, ce qui signifie qu'une classe ne doit avoir qu'un seul emploi. »

Avantages : améliore la lisibilité, la testabilité et la maintenabilité du code. Le code est plus focus et concis.

Pour ce concept, on peut prendre l'exemple d'une université avec des étudiants et chacun d'entre eux possède une carte avec un numéro unique. Pour respecter le principe SRP il faudrait que les étudiants aient comme attribut une carte, mais n'ait pas ses caractéristiques (le numéro unique des cartes) cela permet, de mieux géré, la maintenabilité des cartes et des étudiants dans le programme informatique. Si le système de carte doit avoir des fonctionnalités supplémentaires, il suffit de les modifier dans la classe « carte ».

2) Open Closed Principle

« Les objets ou entités doivent être ouverts pour les extensions mais fermés pour les modifications. »

Avantages : ouvert aux extensions, fermé aux modifications.

Les entités logicielles peuvent être modifiées par extension. Les développeurs n'ont donc pas besoin de redévelopper des choses qui existent déjà. De plus, seules les nouvelles modifications devront être déployées, il n'est pas nécessaire de redéployer complètement le produit dans l'environnement de production.

Pour le concept OCP prenons un exemple d'entreprise avec des développeurs seniors et juniors. N'ayant pas le même niveau de qualification, ils n'ont pas le même salaire. La première idée pourrait être de créer une classe développeur avec un attribut « level », ayant pour valeur junior ou senior. Ainsi un classe « Calculateur de salaire » regarderai l'attribut « level » pour calculer le salaire. En appliquant le principe OCP à cette situation, il suffit de séparer les juniors des seniors en deux classes héritant d'une classe abstraite « développeur ». Ainsi le calculateur de salaire va prendre en compte le type de salaire.

3) Liskov Substitution Principle

Avantages : réduit la dépendance et encourage la réutilisation de code

Imaginons la situation ou nous avons deux machines avec l'une basique et l'autre premium et donc avec des fonctionalité supplémentaires. Or au lieu de créer deux classes différentes pour les séparer on peut créer une interface qui regroupe les méthodes communes aux deux classes. On a donc des classes plus légères pour chacune machine qui conserve leur spécificité

4) Interface Segregation Principle

Avantages : flexibilité, testabilité, réduction du couplage

Le codage par rapport à une interface offre des avantages tels que la flexibilité, la testabilité et la réduction de couplage.

Pennons un exemple, une usine comprend la fois des ouvriers et des robots, les premiers doivent travailler et prendre une pause pour déjeuner tandis que les seconds ne font que travailler. Pour mutualiser la méthode « travail », on peut créer une interface qui implémente cette méthode. Pour les ouvriers qui mangent, on créer une interface « manger ». Quand on développe le programme, on va ainsi créer une classe « robot » qui implémente l'interface « travail » et une classe « ouvrier » qui implémente les interfaces « travail » et « manger ». Pour regrouper les deux classes, robot et ouvrier ont créé aussi une classe travailleuse qui implémentera les interfaces « travail » et « manger ». Nous avons donc une gestion par des interfaces des différentes classes, ce qui permet de mieux distribuer les méthodes.

5) Dependency Inversion Principle

Avantages : Ce concept permet de réduire le couplage des composants sans introduire des modèles de code supplémentaires, ce qui conduit à avoir un schéma d'interaction plus léger et moins dépendant de l'implémentation.

En reprenant l'exemple ci-dessus des machines basiques et premium et appliquant le concept DIP on crée en plus de l'interface qui regroupe les méthodes communes une autre interface qui regroupe les méthodes spécifiques à la machine premium. Cela permet de bien différencier dans le code les différentes fonctionnalités et de changer ou d'ajouter plus facilement des fonctionnalités.