420KBB – TP00

Votre tâche pour ce premier (petit) `TP` sera de concevoir, coder, documenter et livrer un certain nombre de petits éléments. *Portez attention aux consignes*.

Idée générale

Il arrive fréquemment, dans un système informatique, que nous ayons besoin de générer des identifiants uniques. Pensons par exemple à des messages circulant entre deux homologues sur un lien réseau : si plusieurs messages circulent entre les deux pendant une période donnée, par exemple des requêtes et des réponses, il est important de pouvoir distinguer chacun d'eux, entre autres pour être capable de savoir à quelle requête correspond chaque réponse.

Il existe plusieurs stratégies de génération d'identifiants uniques, et il est possible de représenter ces stratégies par des classes. Si l'ensemble de ces classes offre une gamme de services de même signature, alors il est possible de toutes les utiliser à travers une interface commune, ce qui facilite l'entretien du code et réduit le couplage entre code serveur (les générateurs) et code client (le programme consommateur.

Forme du travail

Votre travail prendra la forme d'un système fait de deux unités compilées :

- Une unité « serveur », qui contiendra les classes, interfaces et énumérations ci-dessous, dans un espace nommé `GénérateurId`. Vous devrez faire de cette solution une `DLL` (dans Visual Studio, choisissez un projet de type « Bibliothèque de classes »), donc une unité de code susceptible d'être utilisée dynamiquement par un programme
- Une unité « client », qui fera des tests sommaires pour démontrer que le tout fonctionne correctement. Vous devez faire de cette solution une application console.

Vos chics profs vous fourniront un exemple simpliste de code client, à titre d'illustration, et utiliseront un programme plus complet pour tester votre travail; vous avez donc avantage à écrire un programme de test rigoureux au préalable pour éviter de perdre des points!

Rappel : nous utiliserons cette session C# `12` et la plateforme `.NET 8` alors assurez-vous de bien choisir ces outils en créant vos projets (si vous choisissez `.NET 8` alors vous aurez C# `12` par défaut)!

Modalités de remise

Organisation :	Travail individuel
Date de remise :	**Vendredi le 6 septembre 2024 à 23 h 59**
Code source imprimé?	Oui
Remise par Colnet?	Oui, dans une archive zip nommée[1] comme suit : `Groupe-NomPrénom-TP00.zip` (p. ex. : `05-TromblonGaetan-TP00.zip`) Cette archive doit contenir votre projet serveur **une fois celui-ci nettoyé** (demandez à votre professeur si vous ne comprenez pas cette partie de la consigne)

[1] http://h-deb.clg.qc.ca/CLG/Cours/demander-aide.html#remise_travaux

Code à rédiger côté serveur

Côté serveur, vous serez appelés à rédiger quelques classes, interfaces et énumérations :

- Une classe `DomaineIdentifiants`, qui modélisera des détails de la logique d'affaire qui ne sont pas spécifiques à un générateur ou à l'autre
- Une classe `Identifiant`, dont chaque instance modélisera un identifiant
- Une interface `IGénérateurId`, qui sera implémentée par quelques générateurs, chacun avec sa logique propre
- Les générateurs en tant que tels (détails plus bas)
- Une classe nommée `FabriqueGénérateurs` qui implémentera la logique de fabrication des générateurs en tant que telle

Vous pouvez ajouter des classes au besoin pour vos propres fins, mais le minimum à livrer est ce qui est décrit dans le présent document.

Classe `DomaineIdentifiants`

La logique d'affaires des divers générateurs comprend le respect des bornes de validité des identifiants. Pour cette raison, vous devrez livrer une classe `DomaineIdentifiants`, dont tous les services seront des services de classe (qualifiés `static`). Ces services seront :

- Une propriété de classe `Min`, qui retournera la plus petite valeur possible pour un identifiant
- Une propriété de classe `Max`, qui retournera la plus grande valeur possible pour un identifiant
- Une méthode de classe `Formater`, qui prendra en paramètre la valeur entière d'un identifiant et retournera une chaîne formatée de taille fixe où la valeur de l'identifiant sera précédée par des zéros. **Indice** : la méthode `string.Format` peut vous être utile…

Pour bien comprendre `DomaineIdentifiants.Formater`, deux exemples :

- Si `DomaineIdentifiants.Min==1`, et
- Si `DomaineIdentifiants.Max==100`, alors
- `DomaineIdentifiants.Formater(3)` devra retourner `"003"`, et
- `DomaineIdentifiants.Formater(27)` devra retourner `"027"`.

La longueur de la chaîne retournée devra être la plus courte longueur capable de représenter les valeurs de tous les identifiants possibles.

Note : `DomaineIdentifiants.Formater` n'est pas responsable de valider la valeur passée en paramètre. Que cette valeur soit dans l'intervalle $[Min, Max]$ est une **précondition** de la fonction; dans le cas contraire, la faute revient à l'appelant.

Les valeurs des identifiants doivent être des entiers dans l'intervalle $[0,65535]$. Toutefois, pendant vos tests, il peut être utile de réduire cette plage (p. ex. : utiliser l'intervalle $[1,20]$) pour être en mesure de bien tester la logique de votre code. Bien entendu, `Formater` doit tenir compte des valeurs retournées par `Min` et par `Max` dans son calcul.

Classe `Identifiant`

Vous devrez livrer une classe `Identifiant` exposant au minimum :

- Une propriété `Valeur`, de type `ushort`, publiquement accessible en lecture mais non-modifiable une fois initialisée
- Un constructeur acceptant en paramètre un `ushort` (la valeur de l'identifiant) et une `string` (le préfixe de l'identifiant)
- Un constructeur acceptant en paramètre un `ushort` (la valeur de l'identifiant), représentant le cas où il n'y a pas de préfixe (le cas où le préfixe est une chaîne vide)
- Une méthode `ToString`, qui retournera une `string` débutant par le préfixe de l'identifiant (qui peut être vide) suivi de la valeur de l'identifiant telle que formatée par `DomaineIdentifiants.Formater`

Par exemple, supposant que les identifiants soient dans l'intervalle [0,9999], le code suivant :

```
var id = new Identifiant(3, "ABC");
Console.Write(id);
```

… affichera `ABC0003` à la console.

Interface `IGénérateurId`

Vous devrez livrer une interface `IGénérateurId` exposant au minimum :

- Un service (une méthode d'instance) nommé `Prendre` ne prenant pas de paramètre et retournant le prochain identifiant unique du serveur
- Un autre service nommé `Rendre` prenant en paramètre un identifiant déjà retourné et permettant, selon les implémentations de l'interface, de le recycler

Nos générateurs utiliseront des `ushort` pour représenter les valeurs des identifiants, mais utiliseront le type `Identifiant` en tant que type de retour de `Prendre` et en tant que type du paramètre passé à `Rendre`.

Trois situations problématiques sont possibles :

- Il se peut que le code client essaie de prendre un identifiant d'un générateur alors qu'il n'en reste plus dans celui-ci. Dans ce cas, vous devrez lever un `BanqueVideException` (type de votre cru)
- Il se peut que le code client essaie de rendre à un générateur un identifiant qui n'a jamais été distribué. Dans ce cas, vous devrez lever un `JamaisPrisException` (type de votre cru)
- Il se peut que le code client essaie de rendre à un générateur un identifiant qui a déjà été remis et n'a pas encore été redonné. Dans ce cas, vous devrez lever un `DéjàRenduException` (type de votre cru)

Les générateurs

L'interface `IGénérateurId` devra être implémentée au moins par les quatre classes ci-dessous, toutes inconnues du code client (vous pouvez ajouter des implémentations de votre cru dans la mesure où elles sont cohérentes avec le reste du système).

Chaque générateur prendra en paramètre à la construction une `string`; cette `string` sera potentiellement vide, et servira de préfixe pour les identifiants qu'il générera.

Dans tous les cas, si un générateur ne parvient plus à générer un identifiant unique, sa méthode `Prendre` doit le signaler en levant une exception appropriée.

Les générateurs à écrire sont :

- Une classe `GénérateurSéquentiel` qui génère des identifiants dont les valeurs iront de `DomaineIdentifiants.Min` à `DomaineIdentifiants.Max`, inclusivement et successivement. Une fois que l'identifiant dont la valeur est la valeur maximale possible aura été livré, `Prendre` doit lever une exception signalant cet épuisement. Cette implémentation ne recycle pas les identifiants remis (lorsqu'on lui remet un identifiant, sa seule responsabilité est de signaler s'il s'agit d'un identifiant jamais donné)
- Une classe `GénérateurRecycleur` qui génère des identifiants dans l'ordre, à la manière du `GénérateurSéquentiel`, mais recycle lors d'un appel à `Rendre` les valeurs des identifiants dans une pile de `ushort`. `Prendre` doit offrir l'identifiant dont la valeur est la première sur la pile et ne générer un nouvel identifiant que si la pile est vide
- Une classe `GénérateurAléatoire` qui génère des identifiants dont les valeurs sont choisies de manière aléatoire, et se souvient de ceux déjà générés pour ne pas les offrir une nouvelle fois. Lorsqu'un identifiant a été remis, il redevient sujet à être livré (exprimé autrement, le générateur peut oublier qu'il l'avait donné)
- Une classe `GénérateurPartagé`, qui implémente la même logique que le `GénérateurAléatoire`, mais est telle que tous les clients partageront le même générateur; c'est le rôle de la fabrique (ci-dessous) de s'en assurer

Note : dans le cas du `GénérateurAléatoire` et dans le cas du `GénérateurPartagé`, il sera possible de passer à la construction un germe pour initialiser le générateur de nombres pseudoaléatoires sous-jacent. Ceci permettra de réaliser des tests répétables.

Note : il y aura donc entre zéro et une instance de `GénérateurPartagé` par fabrique (donc s'il y a plusieurs fabriques, chacune pourra avoir son propre `GénérateurPartagé`). Le préfixe d'un `GénérateurPartagé` sera fixé au moment de sa construction. Autrement dit (voir plus bas pour `FabriqueGénérateurs` et `TypeGénérateur`) :

```
var fab = new FabriqueGénérateurs();
// crée un GénérateurPartagé, préfixe "allo", germe 3
var g0 = fab.Créer(TypeGénérateur.Partagé, "allo", 3);
// partage le GénérateurPartagé avec g0, le préfixe demeure "allo"
// (on ne tient pas compte du "coucou")
var g1 = fab.Créer(TypeGénérateur.Partagé, "coucou");
if(g0 != g1) Console.WriteLine("Erreur"); // devraient référer au même objet
```

Classe ***FabriqueGénérateurs***

Puisque nous implémenterons trois types de générateurs, nous les identifierons à l'aide d'une énumération nommée `TypeGénérateur`, qui pourra prendre quatre valeurs symboliques nommées `Séquentiel`, `Recycleur`, `Aléatoire` et `Partagé`.

La fabrication de générateurs devra passer par une fabrique nommée `FabriqueGénérateurs`. Chaque instance de `FabriqueGénérateurs` exposera quatre services :

- Une méthode `Créer` prenant en paramètre un `TypeGénérateur` et retournant un `IGénérateurId` pointant sur le générateur créé. Le préfixe des identifiants générés par ce générateur sera une chaîne vide
- Une méthode `Créer` prenant en paramètre un `TypeGénérateur` et un préfixe (une `string`), retournant un `IGénérateurId` pointant sur le générateur créé. Le préfixe des identifiants générés par ce générateur sera le préfixe qu'il aura reçu à la construction dans le cadre de cette version de `Créer`
- Une méthode `Créer` prenant en paramètre un `TypeGénérateur`, un préfixe (une `string`) et un germe, retournant un `IGénérateurId` pointant sur le générateur créé. Le préfixe des identifiants générés par ce générateur sera le préfixe qu'il aura reçu à la construction dans le cadre de cette version de `Créer`
- Une méthode `ObtenirStatistiques` retournant un tableau de paires faites d'un `TypeGénérateur` et d'un `int`, qui permettra au code client de savoir combien de générateurs de chaque type ont été fabriqués par cette fabrique

Par « tableau de paires », on entend un tableau d'un type immuable contenant deux propriétés et un constructeur.

Si un paramètre dont la valeur ne correspond pas à l'identifiant de l'un des générateurs est passé à `Créer`, cette méthode doit lever un `ArgumentException` standard du cadriciel `.NET`. Ceci inclut le cas où un germe serait passé pour demander la création d'un générateur qui ne serait ne un `GénérateurAléatoire`, ni un `GénérateurPartagé`.

Amusez-vous bien!