Principes SOLID, Design Pattern, TDD

WORKSHOP ETUDIANT

TEXTE DU PROSIT

Dans un pays lointain, un Exar et son tuteur discutent paisiblement.

Maître Tuteur : Alors jeune apprentis, tu as choisi de te spécialiser en développement ?

Etudiant : Oui, mais je sais déjà coder, je ne vois pas ce que je vais apprendre de plus…

Maître Tuteur : Tu es certain de cela ? Oui tu saurais à peu près tout programmer, mais déjà ça te prends toujours un temps fou, et tu réécris souvent une 10aine de fois ton code avant d'arriver à quelque chose de stable...

Etudiant : Heu… nooooooonn…..

Maître Tuteur : Un ingénieur ça ne fait pas comme ça ! Un ingénieur ça ne construit pas un pont en bois, pour qu'il s'effondre au premier passage, pour ensuite se dire « ah tiens j'aurais dû le faire en pierre ». Tu imagines les conséquences ?

Etudiant : Oui bon d'accord mais bon on fait de l'informatique nous….

Maître Tuteur : Y'a pas de mais. Ingénieur ça vient du mot ingénieux, ça signifie être inventif et trouver des solutions, pas faire au petit bonheur la chance ! Un ingénieur ça calcule, ça conçoit, ça doit être en mesure de prédire si la solution sera appropriée ou non.

Etudiant : Oui bon bah concevoir je sais pas faire, mais je sais coder correctement…

Maître Tuteur : Ah bon ? Tu crois que tu serais de taille à faire un développement mais comme un ingénieur ? Tu te sent capable de coder un programme de A à Z sans le lancer une seule fois, et pourtant être certain qu'il fonctionne ?

Etudiant : Heu…. *gloups*

Maître Tuteur : Et bah ok, allons-y ! On va commencer par installer quelques outils : il te faudra un logiciel de modélisation UML, et bien entendu Visual Studio. Ensuite je vais t'apprendre les 5 grands principes SOLID de la qualité logicielle, et bien entendu tu verras les différents Design Pattern. Tu ferras aussi des tests en TDD. Le chemin sera long et sinueux, tu devras venir à bout des 6 épreuves que je te propose, et tu n'as que 2 jours et demi pour le faire. Et en plus tu ne devras ô grand jamais lancer ton programme : plus tu as confiance dans la qualité de ton code, et plus tu pourras attendre jusqu'à ce moment fatidique…

Etudiant : Ça sent la grosse charge de travail ça… Bon, dernière pause cigarette avant… longtemps…

Principes SOLID, Design Pattern, TDD

WORKSHOP ETUDIANT

RESSOURCES

Qualité du Code

https://fr.wikipedia.org/wiki/SOLID_%28informatique%29 Les principes SOLID

https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design Exemples SOLID

http://www.regismedina.com/articles/fr/principes-avances-conception-objet Approfondissement

Design Patterns

https://fr.wikipedia.org/wiki/Patron_de_conception Design Pattern

https://fr.wikipedia.org/wiki/Antipattern Exemples de mauvaises conceptions

Modélisation UML

https://fr.wikipedia.org/wiki/UML_%28informatique%29 Définition

http://uml.free.fr/ Exhaustif mais pas complétement UML 2.0

http://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes Diagramme de Classe

Test Driven Developpement

https://msdn.microsoft.com/en-us/library/hh212233.aspx TDD en C#

https://msdn.microsoft.com/fr-fr/library/ms182532.aspx Implémenter des tests unitaires en C#

stackoverflow.com/questions/933613/how-do-i-use-assert-to-verify-that-an-exception-has-been-thrown

https://msdn.microsoft.com/fr-fr/library/hh270865.aspx Utiliser l'explorateur de test de Visual Studio

Bibliographie

Mémento UML 2.4, de Eyrolles

Design Patterns: Elements of Reusable Object-Oriented Software, Edition : Addison Wesley

Design patterns, collection Tête la première, Edition : O'Reilly

Principes SOLID, Design Pattern, TDD

WORKSHOP ETUDIANT

PREMIERE EPREUVE – LA FORGE LOGICIELLE

Bon alors senseï, on s'y met ?

Bien jeune apprentis codeur, tu sembles avoir hâte de commencer. Le but des différentes épreuves tu l'as compris c'est de faire bien du 1er coup, en lançant ton application au tout dernier moment.

Pour ce faire, nous allons réfléchir à la construction de l'application avant de coder, et faire des Tests Unitaires.

Pourquoi on se prend autant la tête ? Ça prend 10 fois plus de temps (et en plus en stage ils font jamais comme ça et bla bla) …

Ça ne sert pas tout le temps, mais quand on veut faire du code durable et résistant, on est obligé de le faire bien.

Tous les ingénieurs font comme ça, ils réfléchissent avant d'agir et conçoivent leurs systèmes sur papier avant de les réaliser en vrai.

Nous allons donc modéliser notre application…

Oh non ça sent l'UML ça…

Et oui petite salamandre !

Les outils c'est important, chaque forgeron a son marteau. Il te faudra donc à toi aussi tes outils.

Nous allons commencer par ça…

Pour commencer :

1. Installer Visual Studio (la version *Community* fait très bien l'affaire)
2. Installer un logiciel de modélisation UML

   Gratuits : StarUML Modelio UMLet

Principes SOLID, Design Pattern, TDD

WORKSHOP ETUDIANT

DEUXIEME EPREUVE – LE PREMIER PRINCIPE

Tu es prêt ? Bien, ainsi tu veux découvrir comment concevoir un programme de haute qualité ?

Comme je te l'ai dit, il faut dans un premier temps étudier les principes SOLID, et bien connaitre les Design Patterns.

J'ai rien pigé…

C'est normal petit…

Bon, je te donne le premier principe : **La Responsabilité Unique**

Cela signifie que tu dois toujours garder à l'esprit que tes classes doivent faire un minimum de chose. A chaque chose sa classe. Cela te permet de clarifier ton code, et de ne pas avoir à retoucher tes classes sans arrêts. En gros, une classe ne devrait avoir qu'une seule raison de changer.

Ah oui donc pas comme quand je fais une classe *main* avec 200 fonctions ??

* soupire *

Oui exactement… Bon allez, tu peux commencer à coder. Je vais t'expliquer ce que tu dois faire. Et n'oublie pas que tu n'as pas le droit de lancer ton code !

Nous allons recréer l'univers, rien que ça

Design patterns à étudier : Bridge

Pour commencer :

1. Créer une solution, avec un premier projet de type console
2. Ajouter la référence du framework .Net appelée **WindowsBase**
3. Créer un autre projet appelé UnitTestProject de type « test unitaire » dans la solution, et importez la mêmes référence. Importez également le premier projet créé (l'application console) comme référence.
4. Si elle n'est pas présente, ajouter cette référence du framework .Net au projet de test : **Microsoft.VisualStudio.QualityTools.UnitTestFramework**
5. Créer deux packages appelés **Elements** et **Univers** dans l'application console

Principes SOLID, Design Pattern, TDD

WORKSHOP ETUDIANT

Objectif de la mission :

Nous souhaitons réaliser une application qui va permettre de modéliser un **univers** à deux dimensions (X, Y) contenant des **éléments** chimiques. Les éléments pourront être ajoutés dans l'univers, et se verront dotés d'une position. L'univers aura des dimensions finies, elles devront être spécifiées. Vous traiterez toutes les incohérences par des exceptions (éléments en doubles dans l'univers, éléments à la même position, éléments en dehors de l'univers). Il sera possible de parcourir l'univers pour récupérer l'ensemble des éléments, ou bien de connaitre l'élément se trouvant à une certaine position.

Les éléments auront trois propriétés : un nom, un symbole chimique, et une masse. Vous commencerez par un petit échantillon d'**atomes** :

Atome	Symbole	Masse atomique
Carbone	C	12,01074 u
Hydrogène	H	1,00794 u
Lithium	Li	6,941 u

Vous commencerez par modéliser l'application, en UML. Exercez-vous à maitriser la notation du diagramme de classe, et faite la modélisation de vos classes. Les classes qui concernent l'univers et les éléments doivent être dans deux packages séparés.

Vous implémenterez ensuite les tests unitaires. Vous testerez l'ensemble des getters et des setters des propriétés, ainsi que le déplacement. Vous n'oublierez pas de tester les exceptions.

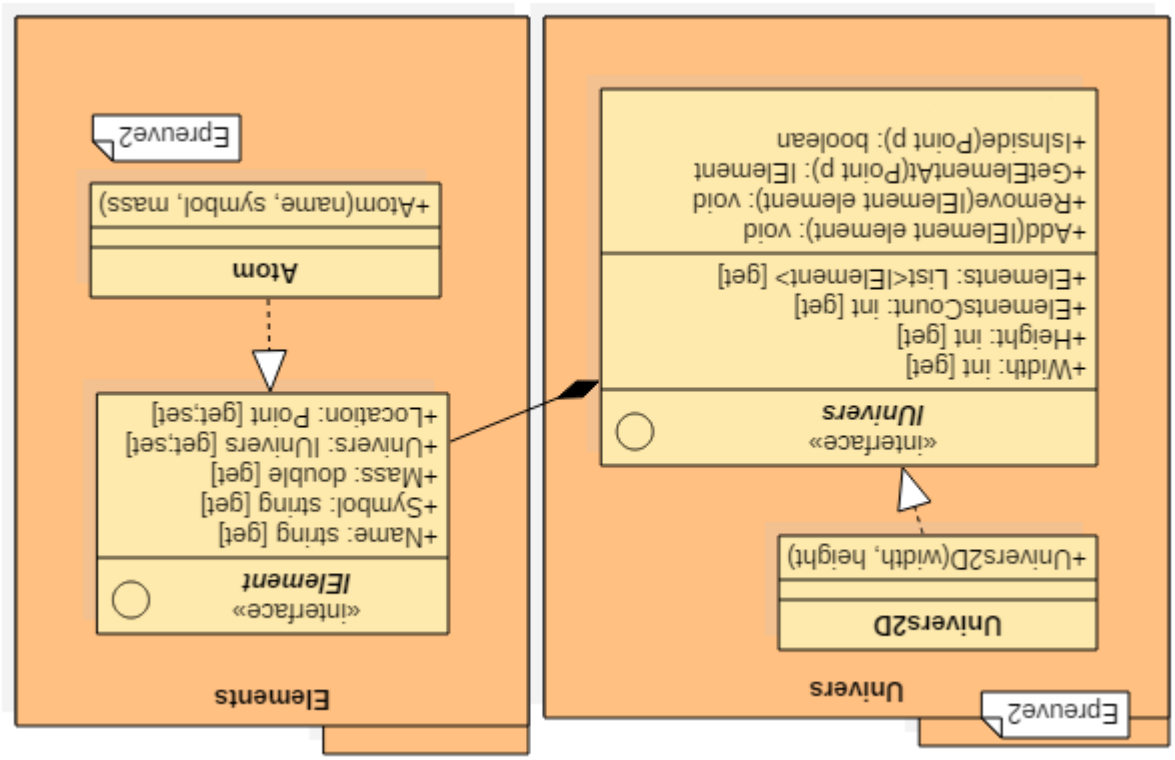
Vous tenterez au maximum d'implémenter les tests en premiers. Exercez-vous à prévoir les situations pouvant générer des erreurs, et testez les toutes. Ensuite, vous implémenterez les classes des atomes et de l'univers.

Voilà comment vous procéderez :

1. Modéliser l'application sous forme de diagramme de classe. Toute l'application doit être modélisée.
2. Envisagez au maximum le changement, prenez bien soin de décomposer vos abstractions de vos implémentations (Bridge), ça vous servira par la suite !
3. Créer vos tests dans le projet prévu à cet effet.
4. Utiliser la vue « Explorateur de test » dans Visual Studio. N'oubliez pas que vos méthodes de test doivent être publiques pour s'afficher. Si vous avez bien annoté vos classes et les méthodes, elles devraient toutes apparaitre dans cette fenêtre.
5. Vous n'avez pas le droit de lancer votre programme. Vous ne devrez exécuter que les tests en utilisant le bouton « **Exécuter tout** » de la fenêtre « Explorateur de test ».

SOLUTION

Voici un exemple d'implémentation de cette partie. Note : je mets dans ce schéma des attributs aux interfaces, car j'utilise des propriétés getter/setter en C# qui sont en fait des méthodes déguisées.



Exemple de test :

```
[TestMethod]
public void Test_Elements_Creation()
{
	var a1 = new ChemicalElement("Carbone", "C", 12.01074);
	var a2 = new ChemicalElement("Hydrogen", "H", 1.00794);
	Assert.AreEqual("C", a1.Symbol);
	Assert.AreEqual(1.00794, a2.Mass);
}
```

Principes SOLID, Design Pattern, TDD

WORKSHOP ETUDIANT

TROISIEME EPREUVE – LE SECOND PRINCIPE

Alors jeune padawan, tu as terminé ?

Et tu n'as pas touché à ton main j'espère?

Oui je crois que j'ai terminé, et non je n'ai pas lancé mon programme je suis un padawan assidu et sérieux, il ne me viendrait jamais à l'idée de …

Oui bon bref ! Et j'espère que tu n'as pas créé de classes pour chaque composant chimique ? Sinon il faudrait créer une classe à chaque fois et il y a 118 éléments dans l'univers !

Sinon, je vais te dire le deuxième principe : **Principe Ouvert/Fermé.** « Ouverte » signifie qu'une classe a la capacité d'être étendue. « Fermée » signifie qu'elle ne peut être modifiée que par extension, sans modification de son code source.

Ça me fait une belle jambe…

Espèce de cornichon albigeois, j'essaye de t'expliquer qu'il faut prévoir le changement. Une application ça évolue. Ce principe énonce qu'il faut anticiper les modifications futures, et justement il existe des Design Patterns pour ça…

Design patterns à étudier : Factory, Observer, Singleton

Objectif de la mission :

Vous allez maintenant **déporter l'instanciation des éléments** (le mot clé *new*) dans une fabrique. La fabrique sera accessible grâce à un singleton, et permettra de créer des éléments préconfigurés (nom, masse, symbole) à partir du simple nom de l'élément chimique.

Vous allez aussi ajouter des événements qui seront levés :

- Sur l'univers lors de l'ajout d'un élément dans l'univers
- Sur l'univers lors du retrait d'un élément de l'univers
- Sur les éléments lors du déplacement spatial

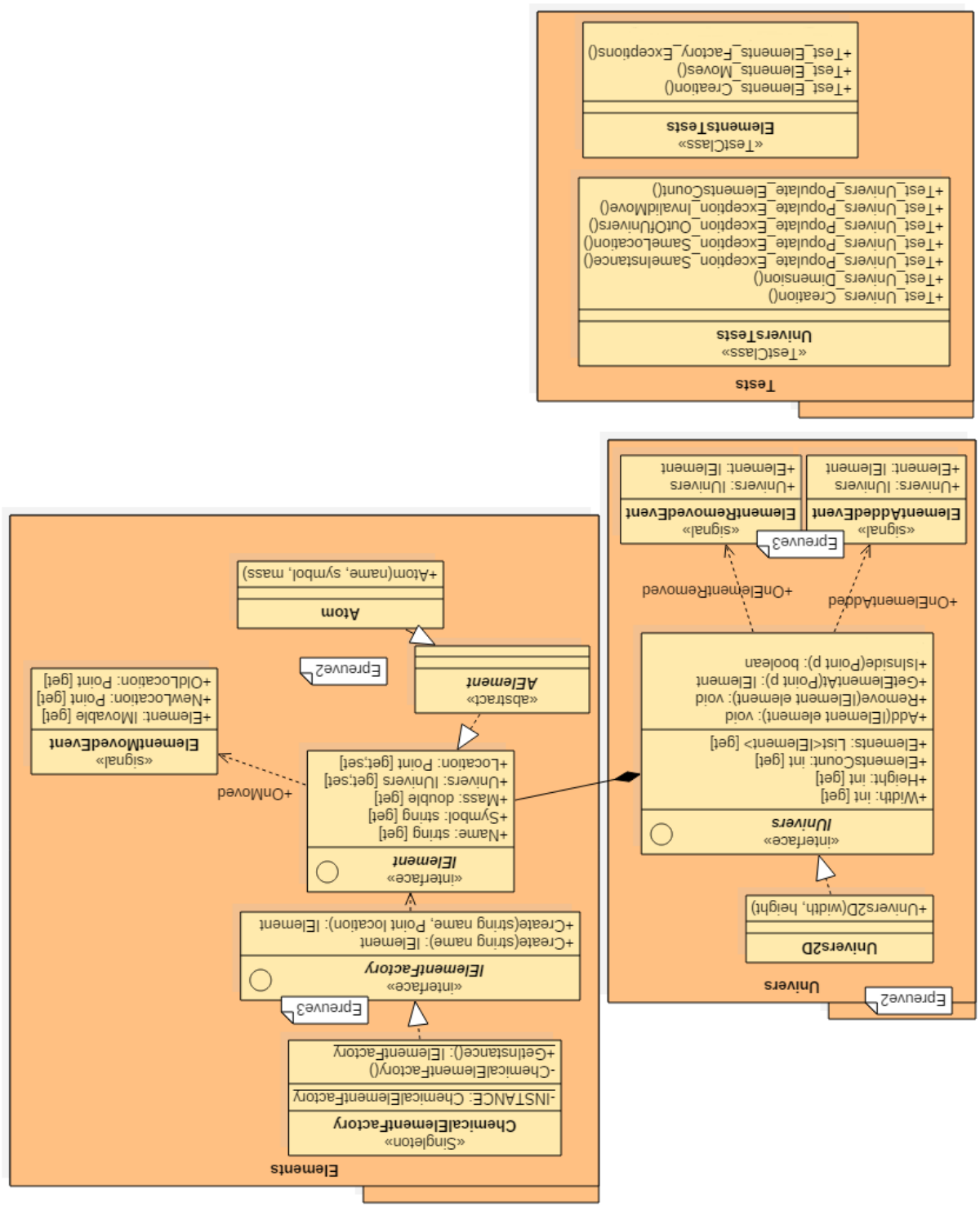
Voilà comment vous procéderez :

1. Implémenter la fabrique abstraite (en Singleton) et le DP Observer (3 events)
2. Mettez à jour la modélisation
3. Tester votre fabrique, et ses exceptions

SOLUTION

Voici un exemple d'implémentation de cette partie. Note : je mets dans ce schéma des attributs aux interfaces, car j'utilise des propriétés getter/setter en C# qui sont en fait des méthodes déguisées.

De plus, en C# les *events* peuvent être déclarés dans les interfaces également.



QUATRIEME EPREUVE – LE TROISIEME PRINCIPE

Ok done ! Et avant que vous demandiez : non je n'ai pas lancé mon *main* !

Ah c'est bien jeune batracien, tu apprends vite ! Constate que maintenant, ton code pourra facilement être étendu, par exemple en ajoutant des *listeners* qui pourront rajouter des fonctionnalités.

J'espère que tu as aussi remarqué que le fait d'exporter les instanciations permet surtout de faire en sorte que ton code ne dépende plus que des abstractions, et que toutes les implémentations se retrouvent dans la fabrique.

Oui j'ai remarqué. Mais dites, vous n'êtes pas sensé m'apprendre un nouveau principe à chaque fois ?

Exacte petit rhododendron !

Voici le troisième principe SOLID : la **Substitution de Liskof !**

L'idée, c'est que des sous-types qui héritent d'une classe doivent obligatoirement être substituables, c'est-à-dire qu'utiliser une sous-classe doit se faire de la même manière que la classe d'origine.

Oula, il a l'air encore pire que les autres celui-là…

En fait non, mais ce principe vise surtout à éviter de provoquer des effets de bords quand on surcharge du code existant.

La mission maintenant permet de se passer d'héritage pour surcharger un type, mais le principe s'applique tout autant !

Objectif de la mission :

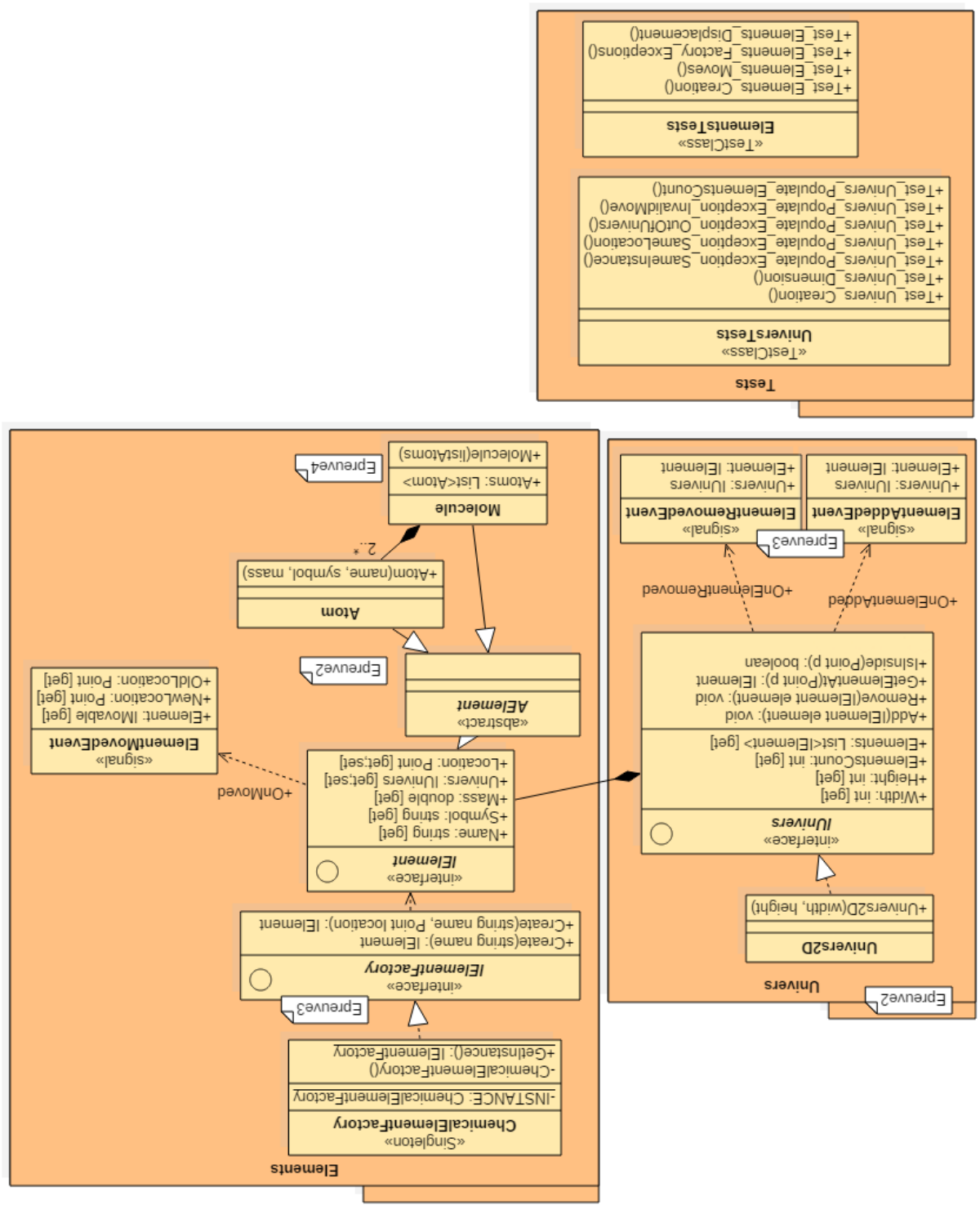
Vous allez maintenant ajouter un autre type d'élément dans l'univers : les **molécules**.

Les molécules sont formées à partir d'un assemblage d'au moins deux atomes. La masse des molécules est égale à la somme de l'ensemble de ses atomes. Quant au symbole de ces éléments, ils sont composés à partir de leur composition atomique. Ainsi, l'eau est composée de 2 atomes d'hydrogène et d'un atome d'oxygène, sa formule est donc « H2O ». Si vous êtes motivé, vous pouvez implémenter aussi le nom des molécules (« Oxyde de dihydrogène » pour l'eau).

Assurez de rendre la création des atomes la plus simple possible : on entend par là que le constructeur doit comporter le nombre d'argument le plus réduit possible. N'oubliez pas qu'au sein de l'univers, vos atomes et vos molécules doivent être parfaitement substituables en comportement.

SOLUTION

Voici un exemple d'implémentation de cette partie. Note : je mets dans ce schéma des attributs aux interfaces, car j'utilise des propriétés getter/setter en C# qui sont en fait des méthodes déguisées.



CINQUIEME EPREUVE – LE QUATRIEME PRINCIPE

Bien jeune palmipède, tu as terminé ?

J'espère que cela ne t'a pas donné trop de travail… Mais tu avais peut-être anticipé l'arrivée d'un nouveau type d'élément ?

…

C'était nécessaire pour que tu retiennes : le pire ennemi du développeur c'est le **changement**. Si on bâtit bien ses constructions en réfléchissant, on peut l'anticiper et lui résister.

Bon, passons au quatrième principe SOLID : la **Ségrégation des Interfaces** ! Le principe, c'est de bien séparer en différentes interfaces les comportements d'une classe. Ainsi, les autres classes qui vont utiliser ce code ne doivent pas dépendre de plus de ce dont elles ont besoin.

Hum je crois que je commence à comprendre… ça peut également servir quand on cherche à se répartir le travail à plusieurs en projet, chacun utilise les interfaces des autres ?

Exactement jeune pousse macrobiotique ! Passons à la pratique…

Design patterns à étudier : Stratégie

Objectif de la mission : Nous allons bientôt afficher notre univers, mais nous allons également tenter de coder le **déplacement des éléments** avant de lancer le programme.

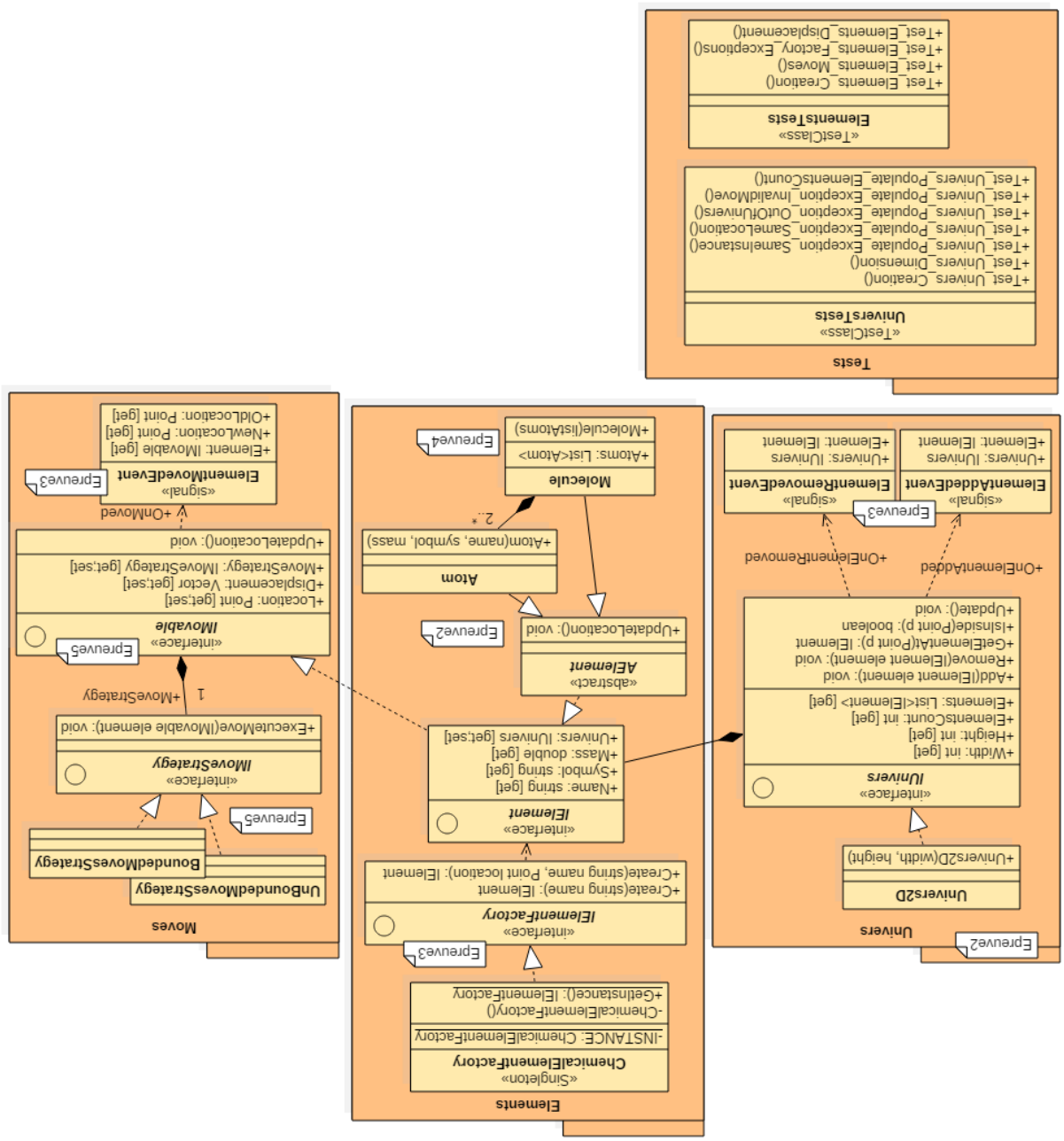
Pour cela, vous allez extraire de l'interface des éléments tout ce qui attrait au déplacement (propriétés, méthodes, events …) dans une autre interface *IMovable*. Vous allez faire un peu de refactoring pour isoler tout le comportement de déplacement. Ajoutez ensuite à l'interface créée une propriété pour matérialiser la vitesse de déplacement de l'élément dans l'espace, et une méthode *UpdateLocation()* ne retournant rien.

Vous ajouterez ensuite une *stratégie* de déplacement (qui dépendra de votre nouvelle interface uniquement). Cette stratégie sera appelé par la méthode *UpdateLocation()* de l'élément, et déplacera l'élément dans l'univers en prenant en compte son vecteur de déplacement. Quand les éléments atteindront les limites de l'univers, le vecteur sera inversé pour qu'ils ne puissent en sortir. Vous ferez ensuite une autre stratégie qui ne se bornera pas aux limites de l'univers : elle ferra réapparaitre les éléments à droite dès qu'ils sortent à gauche par exemple.

Bien entendu, ajouter les tests unitaires liés au déplacement avant de code l'implémentation. Si votre code est correct, vous lèverez une exception quand une collision se produit. Faites en sorte d'en faire un événement, qui se déclenchera sur l'élément.

SOLUTION

Voici un exemple d'implémentation de cette partie. Note : je mets dans ce schéma des attributs aux interfaces, car j'utilise des propriétés getter/setter en C# qui sont en fait des méthodes déguisées.



SIXIEME EPREUVE – LE LANCEMENT !

Bravo petit scarabée ! Tu te débrouilles pas mal…

Regarde, maintenant si on veut que le déplacement se fasse différemment (comme avec de la gravité par exemple) il suffira de rajouter une stratégie, sans modifier le reste du programme !

Ah mais c'est génial ce truc ! On peut même changer de stratégie en pleine exécution !

Et oui, ça simplifie souvent la vie. Utilise la stratégie dès qu'un objet pourrait avoir plusieurs manières de réaliser une capacité. Ou si tu souhaites faire une première version simple de quelque chose, pour ensuite l'améliorer…

Bon. Et bien je pense qu'il est temps de lancer ce programme…

Ahh !!

Objectif de la mission :

Nous allons maintenant rajouter une **représentation visuelle** à notre application console. Faites en sorte d'afficher l'univers et ses éléments dans la console en ASCII, sous la forme d'un tableau qui sera rafraichit. N'oubliez pas le premier principe que nous avons évoqué !

On va ensuite rajouter un moteur, qui donnera vie à l'univers. Ce moteur donnera le tempo, les éléments auront donc effectué des déplacements à chaque pas de temps. On implémentera un moteur avec un Thread, avec une boucle infinie et un temps d'attente entre chaque mise à jour. Ce temps d'attente devra être configurable. Et attention, le moteur de rendu et le moteur de tempo devront uniquement dépendre d'interfaces, selon le quatrième principe SOLID que nous avons évoqué.

Cela devrait vous prendre déjà un bon moment…

Quand vous aurez terminé, il est temps d'écrire le *main* de l'application. Il doit normalement être simplissime :

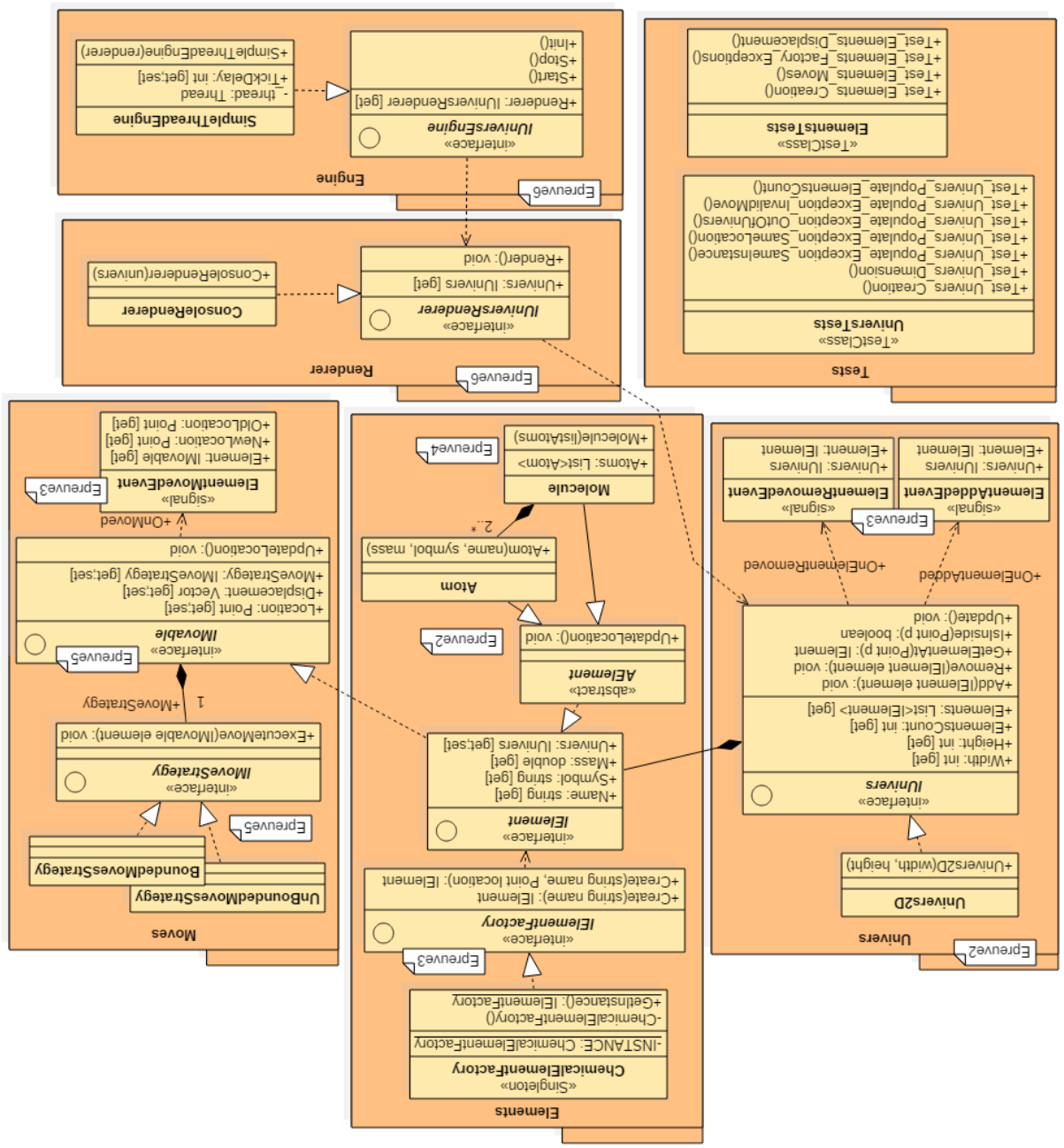
1. Création de l'univers avec ses dimensions
2. Peupler des éléments dans l'univers
3. Création du moteur d'affichage, en lui passant l'univers en paramètre
4. Création du moteur de tempo, en lui passant le moteur d'affichage et le délai
5. Lancement du thread

Allez-y, vous pouvez (enfin) lancer !!

Alors, marchera ou marchera pas du premier coup ? ☺

SOLUTION

Voici un exemple d'implémentation finale. Note : je mets dans ce schéma des attributs aux interfaces, car j'utilise des propriétés getter/setter en C# qui sont en fait des méthodes déguisées.



SEPTIEME EPREUVE – LE CINQUIEME PRINCIPE

Mais heu, dites-moi grand maître tuteur devant l'Absolu… On n'avait pas parlé de 5 principes SOLID ? Si mes calculs sont bons on n'en n'a vu que 4 ?

Exacte jeune Padawan !

Le dernier principe est de loin de plus complexe : il vise à repenser totalement la manière dont on conçoit une application.

C'est **l'Inversion des Dépendances**. Il se formule ainsi :

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend on details. Details should depend on abstractions.

Ah mais vous parlez carrément en anglais maintenant ?

Je t'explique. Traditionnellement, on conçoit les applications pour que les modules de **haut niveau** (ceux qui portent la logique fonctionnelle, le métier) utilisent des composants de **bas niveau** (librairies, couches d'accès aux données ou framework d'IHM). Ainsi, on a une **dépendance** du haut niveau vers les composants de bas niveau.

Le problème, c'est que dès que les composants de bas niveau évoluent et changent de version, on doit obligatoirement impacter le code métier. Et l'application est souvent fortement **couplée** avec sa couche d'IHM.

Le but du dernier principe est d'éviter ce haut niveau de couplage, en utilisant une couche d'abstraction. Cela permet une meilleure réutilisation du code métier.

Mais soyons clair, c'est assez poussé comme paradigme. On va en rester là pour aujourd'hui ! Mais si tu veux t'y intéresser libre à toi.

Heu mais oui bien sûr…

Enfin là si ça vous dérange pas je vais aller me mettre en PLS…

Notions à étudier : DIP, Inversion of control (IoC)

FIN

Bravo jeune padawan !

Tu as vraiment bien travaillé si tu es arrivé jusqu'ici !

Nan mais c'était vachement balèze comme exercice !

Vous n'avez pas idée de me donner autant de travail sur 2 jours et demi ??

Je vais me plaindre à la convention de Genève…

Héhé oui, c'était certainement difficile, mais tu as appris des tonnes de choses et ça en valait la peine !

Bon, si tu dois retenir une chose pour résumer, c'est qu'il est important quand tu programmes de penser à séparer tes interfaces de tes implémentations. Cela permet de réduire ce qu'on appelle le couplage entre tes composants.

Quand tu réduis le couplage, tu fais des classes plus réutilisables, et cela te permet de capitaliser petit à petit. Tu peux ainsi constituer une sorte de *framework* de classes réutilisables qui font seulement un job, mais le font bien.

Et bien entendu, en gardant à l'esprit que tu peux faire *bien* du premier coup, à partir du moment où tout est bien modélisé, essaye d'apprendre à *concevoir* ton application avant de coder. Tu verras, ça change vraiment tout…

Je vous crois grand maitre, votre barbe rassurante et pleine de sagesse m'incite à vous faire confiance…

Félicitations !

Conception : Rémi BELLO (Toulouse)