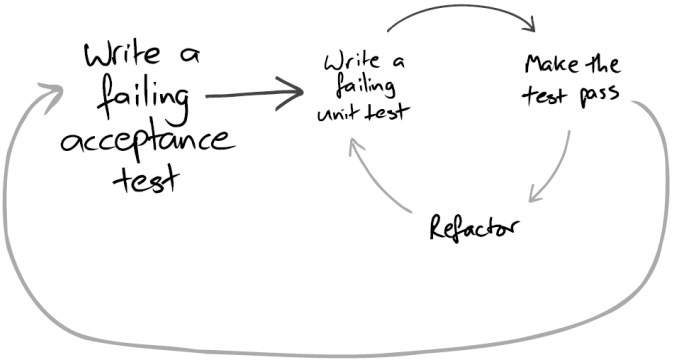
Travaux dirigés 2

Développement piloté par les tests

Le développement piloté par les tests (TDD) est une pratique agile qui entremêle la programmation, l'écriture de tests et le remaniement du code. Le principe fondamental est d'écrire les tests avant d'implémenter le système. Dans cette optique, la création de tests devient une véritable activité de conception, dans laquelle le comportement du système est spécifié sans ambiguïté. La figure suivante résume cette manière de travailler, qui combine deux typologies de test : les *tests d'acceptation*, qui sont des procédures permettant de s'assurer que le système soit conforme aux fonctionnalités souhaitées, et les *tests unitaires*, qui sont des courts programmes servant à vérifier le bon fonctionnement de chaque morceau de code du système.



Plus précisément, le TDD enferme le développement dans un cycle où un test unitaire doit être écrit juste avant d'implémenter le morceau de code du système qui permet de le réussir. Si nous travaillons de cette manière, nous construisons progressivement une suite de tests (unitaires et d'acceptation) pouvant être exécutée à chaque évolution du code du système. Cela est essentiellement l'esprit de l'approche *agile* : plutôt que tenter dès le début de concevoir un système idéale assez générique pour accueillir toutes les fonctionnalités, l'idée est de rechercher une solution simple sans essayer d'aller au-delà, quitte à devoir remanier régulièrement le code déjà existant.

Le présent TD est organisé en trois sections, dont les deux premières proposent des exercices qui visent à faire comprendre les principes fondamentaux du développement piloté par les tests. En revanche, la troisième section porte sur un mini-projet concernant la réalisation du jeu « tetris ».

2.1 Tests unitaires

Les tests unitaires sont des courts programmes qui vérifient le bon fonctionnement de chaque morceau de code du système. Dans le TDD, ils sont utilisés pour enfermer le développement dans un cycle de trois pas :

1. **phase rouge** – créer un test unitaire qui décrit un aspect du système ;
2. **phase verte** – écrire juste assez de code pour réussir le test ;
3. **remaniement** – nettoyer le code existant.

Par exemple, supposons de vouloir coder une méthode qui compte le nombre de chiffres utiles dans un entier non négatif (7 a une chiffre, 14 en a deux, etc). Un premier test porte sur la plus petite valeur, soit le zéro.

```
public void Test0() {
   Assert.AreEqual( CompterChiffres(0), 1 );
}
```

Ce code ne compile pas, car la méthode `CompterChiffres` n'est pas écrite. Pour compiler, on peut se contenter dans un premier temps d'écrire :

```
public static int CompterChiffres(long n) {
   return 1;
}
```

Le test passe. Avant de continuer, on remanie le code de test en écrivant une fonction qui puisse tester n'importe quelle valeur passée en paramètre : cela nous évitera d'avoir à dupliquer du code !

```
void TestValeur(long n, int vrai_resultat) {
   Assert.AreEqual( CompterChiffres(n), vrai_resultat );
}

public void Test0() {
   TestValeur(0, 1);
}
```

On ajout un deuxième test pour le plus grand nombre à une chiffre, soit le 9.

```
public void Test9() {
   TestValeur(9, 1);
}
```

Les deux tests passent. On continue en ajoutant un troisième test pour le plus petit nombre à deux chiffres, soit le 10.

```
public void Test10() {
   TestValeur(10, 2);
}
```

Bien entendu, le test ne passe pas. Il faut coder la méthode `CompterChiffres` de maniéré plus réaliste. On la réécrit avec une itération qui compare la valeur d'entrée `n` aux puissances successives de 10.

```
int CompterChiffres(long n)
{
   int resultat = 1;
   long p10 = 10;
   while( n >= p10 ) {
      p10 *= 10;
      resultat++;
   }
   return resultat;
}
```

Les trois tests passent maintenant.

Exercice 2.1

L'outil de développement « Visual Studio » dispose d'un framework permettant l'exécution automatisée de tests unitaires. Votre tache est d'utiliser cette infrastructure pour mettre en ouvre l'exemple illustré précédemment.

1. Créez un projet « Console Application » et puis un projet « Unit Test Project ». Ajoutez à ce dernier une référence au premier projet.
2. Implémentez un test dans le deuxième projet.

```
[TestClass]
public class CompterChiffresTest
{
   void TestValeur(long n, int vrai_resultat) {
      Assert.AreEqual( CompterChiffres(n), vrai_resultat );
   }

   [TestMethod]
   public void Test0() {
      TestValeur(0, 1);
   }
}
```

3. Ouvrir le *Test Explorer* (à partir du menu « Test ») et lancez les tests.
4. Le test ne compile pas, car `CompterChiffres` n'est pas encore écrite. Implémentez-la dans le premier projet et vérifiez que le test passe.
5. Ajoutez d'autres tests et vérifiez qu'ils passent.
6. Implémentez le test ci-dessous et vérifiez qu'il échoue.

```
[TestMethod, Timeout(1000)]
public void TestEntierMax() {
   TestValeur(Int64.MaxValue, 19);
}
```

7. Trouvez le bug dans la méthode `CompterChiffres`. Pour ce faire, placez un breakpoint dans son implémentation et lancez le test échoué en sélectionnant « Debug Selected Test » dans le *Test Explorer*.
8. Corrigez la méthode `CompterChiffres` afin que tous les tests passent. (*Conseil* : faire décroitre la variable `n` au lieu de faire monter `p10`)

2.1.1 Micro-transformations

Le TDD impose de commencer par écrire les tests unitaires avant le code du système. Toutefois, cette règle ne constitue que la partie visible de l'iceberg. Il est essentiel que le cycle « rouge-vert-remaniement » soit réalisé très rapidement, en quelques minutes tout au plus. C'est pourquoi le TDD est gouverné par les trois lois suivantes.

1. **Première loi** : vous ne devez pas implémenter du code tant que vous n'avez pas écrit un test unitaire qui s'échoue.
2. **Deuxième loi** : vous devez uniquement écrire le test unitaire suffisant pour échouer ; l'impossibilité de compiler est un échec.
3. **Troisième loi** : vous devez uniquement implémenter le code, le plus simple possible, suffisant pour réussir le test courant qui s'échoue.

La particularité du TDD est donc de réaliser des cycles minuscules où le code est progressivement généralisé par l'ajout de nouveaux tests. Dans chaque cycle, nous sommes habilités seulement à réaliser des *micro-transformations*, dont les plus courantes sont énumérées ci-dessous par ordre de complexité. Pour réussir un test, il est préférable de privilégier les transformations à basse complexité, afin d'encourager des cycles courtes.

1. **{}** → **nil** : création d'un code qui produit les plus simple résultat.
2. **Nil** → **constant** : introduction d'une constante.
3. **Constant** → **complex constant** : changement d'une constante.
4. **Constant** → **scalar** : création ou utilisation d'une variable.
5. **Instruction** → **instructions** : ajout d'instructions.
6. **Unconditional** → **if** : division du chemin d'exécution.
7. **Scalar** → **array** : substitution d'une variable simple par un tableau.
8. **Array** → **container** : promotion d'un tableau en un conteneur.
9. **Statement** → **recursion** : emploi de l'approche récursive.
10. **If** → **while/for** : introduction d'une boucle.
11. **Expression** → **function** : morceau de code remplacé par une fonction.
12. **Variable** → **assignment** : modification du contenu d'une variable.

Par exemple, considérons une fonction `Generate` qui calcule les facteurs premiers d'un entier ($10 = 2 \times 5$, $360 = 2^3 \times 3^2 \times 5$, etc). Pour simplifier les tests, nous introduisons d'abord les fonctions suivantes.

```
void TestValue(int n, List<int> expected)
{
   CollectionAssert.AreEqual( PrimeFactor.Generate(n), expected );
}

List<int> L(params int[] v)
{
   return new List<int>(v);
}
```

Le premier test porte sur l'entier 1, qui n'a pas de facteurs premiers. Pour réussir ce test, il suffit de renvoyer une liste vide (transformation #1).

```
public void Test1()
{
  TestValue( 1, L() );
}
```

```
static List<int> Generate(int n)
{
   return new List<int>();
}
```

Le deuxième test porte sur l'entier 2, qui admet le facteur 2. Pour réussir ce test, il suffit d'introduire une structure `if-then` (transformation #6).

```
public void Test2()
{
  TestValue( 2, L(2) );
}
```

```
static List<int> Generate(int n)
{
   List<int> factors = new List<int>();
   if (n > 1)
      factors.Add(n);
   return factors;
}
```

Maintenant, tous les tests sur les nombres premiers passent.

```
public void Test3()
{
   TestValue( 3, L(3) );
}
```

```
public void Test5()
{
   TestValue( 5, L(5) );
}
```

Le prochain test porte sur l'entier 4, qui admet les facteurs $\{2, 2\}$. Pour réussir ce test, on peut vérifier si `n` est divisible par 2 : dans l'affirmative, on ajoute 2 au tableau et on effectue la division de `n` par 2 (transformation #6). Ce faisant, le code marche avec tous les nombres premiers multipliés par 2.

```
public void Test4()
{
  TestValue( 4, L(2,2) );
}

public void Test6()
{
  TestValue( 6, L(2,3) );
}
```

```
static List<int> Generate(int n)
{
   List<int> factors = new List<int>();
   if (n > 1) {
      if(n % 2 == 0) {
         factors.Add(2);
         n /= 2;
      }
   }
   if(n > 1) factors.Add(n);
   return factors;
}
```

Nous commençons à entrevoir la structure de l'algorithme. La prochaine étape est de gérer les puissances de 2. Pour ce faire, il suffit de remplacer le `if` par une boucle (transformation #10). Nous remarquons que cela marche également avec tous les nombres premiers multipliés par une puissance de 2.

```
public void Test8()
{
  TestValue(8, L(2,2,2));
}

public void Test12()
{
  TestValue(12, L(2,2,3));
}
```

```
static List<int> Generate(int n)
{
   List<int> factors = new List<int>();
   if (n > 1)
   {
      for( ; n % 2 == 0 ; n /= 2)
         factors.Add(2);
   }
   if(n > 1) factors.Add(n);
   return factors;
}
```

Nous arrivons au sprint final : généraliser l'algorithme pour les facteurs premiers d'autres que 2. A cet égard, considérons le test sur l'entier 9, qui admet les facteurs $\{3, 3\}$. Pour réussir ce test, nous avons un peu de travail à faire. Nous allons y arriver petit à petit en s'assurant que c'est toujours le même test qui échoue, jusqu'à le faire passer. Commençons par transformer la constante 2 en une variable (transformation #4).

```
public void Test9()
{
  TestValue(9, L(3,3));
}
```

```
static List<int> Generate(int n)
{
   List<int> factors = new List<int>();

   if (n > 1)
   {
      int f = 2;
      for( ; n % f == 0 ; n /= f)
         factors.Add(f);
   }

   if(n > 1)
      factors.Add(n);

   return factors;
}
```

Le test échoue. La raison est simple : nous devons essayer tous les possibles facteurs premiers. Pour ce faire, nous pouvons remplacer le `if` par une boucle (transformation #10) qui incrémente la variable précédemment crée.

```
public void Test9()
{
  TestValue(9, L(3,3));
}

public void Test18()
{
  TestValue(18, L(2,3,3));
}
```

```
static List<int> Generate(int n)
{
   List<int> factors = new List<int>();

   for (int f=2 ; n > 1 ; f++)
   {
      for( ; n % f == 0 ; n /= f)
         factors.Add(f);
   }

   return factors;
}
```

Enfin, le test réussit. Nous concluons le développement par l'écriture de quelques tests d'acceptation, afin de vérifier le bon fonctionnement du code.

```
public void Test360()
{
   TestValue( 360, L(2, 2, 2, 3, 3, 5) );
}

public void Test1001()
{
   TestValue( 1001, L(7, 11, 13) );
}

public void Test1010021()
{
   TestValue( 1010021, L(17, 19, 53, 59) );
}
```

Exercice 2.2

La numération romaine est un système pour représenter des nombres entiers à l'aide de symboles combinés entre eux. Il existe 7 lettres pour composer les nombres : I=1, V=5, X=10, L=50, C=100, D=500 et M=1000. La construction d'un nombre suit les règles suivantes.

— Tout symbole qui suit un autre de valeur supérieure ou égale s'ajoute à celui-ci. Par exemple, II=2, III=3, VI=6, VII=7, VIII=8, etc.

— Les symboles I, X, C et M peuvent être répétés jusqu'à trois, alors que les symboles V, L et D ne peuvent pas être répétés. Par conséquent, IIII, VV, LL, DD, et ainsi de suite, ne sont pas des nombres valides.

— Tout symbole qui précède un autre de valeur supérieure se soustrait à ce dernier. Par exemple, IV=4, IX=9, XL=40, XC=90, CD=400 et CM=900. Toutefois, IL et IC sont interdits, car 49=XLIX et 99=XCIX.

— Les symboles sont groupées par ordre décroissant, sauf pour les valeurs à retrancher. Par exemple, 369=CCCLXIX et 2751=MMDCCLI.

L'objectif de cet exercice est d'implémenter une fonction qui calcule la représentation romaine d'un entier entre 1 et 3999. À présent, vous savez que le TDD préconise de commencer par le test et le code le plus simple possible.

```
void TestValue(int n, string r) {
  Assert.AreEqual(ToRoman(n), r);
}
void Test1() {
  TestValue(1, "I");
}
```

```
public class Converter
{
    static string ToRoman(int n) {
      return null;
    }
}
```

1. Utilisez la transformation #2 pour réussir le premier test. Ajoutez ensuite un test sur le nombre 2 et corrigez le code avec la transformation #6, puis un test sur le nombre 3 entraînant la transformation #10.
2. Ajoutez un test sur le nombre 5 et utilisez la transformation #6 pour réussir ceci. Testez également les nombres 6, 7 et 8.
3. Vous commencez à saisir la structure de l'algorithme. Pour généraliser le code, testez progressivement des nombres composés par les autres symboles, tels que 27, 51, 123, 521 et 2012. Ne testez pas encore les nombres que l'on obtient par soustraction de symboles (4, 9, etc).
4. Avant d'ajouter des nouveaux tests, remarquez s'il y a des répétitions dans votre code. Dans l'affirmative, remaniez-le afin d'éliminer la redondance, an sachant que l'algorithme peut être implémenté par deux boucles imbriqués qui traversent des conteneurs.
5. Ajouterez progressivement les tests sur les nombres 4, 9, 40, 49, 90, 99, 400 et 900. Comment modifier le code afin de gérer les retranches ?
6. Enfin, vérifiez que 369=CCCLXIX et 2751=MMDCCLI.

2.1.2 Schéma Arranger-Agir-Auditer

Un test unitaire doit se focaliser sur un seul concept, afin d'aboutir à une conclusion unique qui est facile à comprendre. Nous ne voulons pas de longues fonctions qui testent diverses choses hétéroclites l'une après l'autre. Nous préférons plutôt *clarté* et *simplicité*. C'est pourquoi un test unitaire est généralement organisé en trois parties, suivant le schéma ci-dessous.

1. **Arranger**. Dans un premier temps, on définit les objets et les variables nécessaires au bon fonctionnement du test.
2. **Agir**. Ensuite, on procède à l'exécution de l'action que l'on souhaite tester (en général, exécuter la méthode que l'on veut tester).
3. **Auditer**. Enfin, on vérifie que le résultat est conforme aux attentes.

Le schéma AAA permet d'écrire un test unitaire en respectant la convention « *étant donné* un contexte, *lorsque* un événement survient, *alors* un résultat se produit ». Ce faisant, quiconque lit le test est capable de déterminer très rapidement son objectif, sans être trompé ou submergé par les détails.

```
void TestTooCold()
{
    // arranger
    Device hw = new Device();
    hw.SetTemp(WAY_TOO_COLD);

    // agir
    hw.Tic();

    // auditer
    Assert.IsTrue (hw.HeaterState());
    Assert.IsTrue (hw.BlowerState());
    Assert.IsFalse(hw.CoolerState());
    Assert.IsFalse(hw.HiTempAlarm());
    Assert.IsTrue (hw.LoTempAlarm());
}
```

L'exemple ci-dessus montre le test d'un système de contrôle de l'environnement. Sans entrer dans les détails, nous pouvons deviner que le test vérifie si l'alarme de température faible, le système de chauffage et le ventilateur sont en marche lorsque la température baisse trop. Bien qu'il y a plusieurs assertions, ce test se focalise sur un seul concept, notamment le fait que l'état final du système soit cohérent avec la température qui baisse trop.

Dans une certaine école de pensée, il est dit que chaque test ne doit inclure qu'une et une seule assertion. Cette règle peut sembler draconienne, mais elle nous encourage à créer un langage de test spécifique au domaine qui la prend en charge. Dans l'exemple précédent, nous pouvons introduire une fonction `GetState` qui convertit l'état du système dans une chaine de caractères. Une lettre majuscule signifie « allumé », une lettre minuscule signifie « éteint », et les lettres sont toujours données dans l'ordre suivant : chauffage (`Heater`), ventilateur (`Blower`), conditionneur (`Cooler`), alarme de température haute (`HiTemp`), alarme de température baisse (`LoTemp`).

```
public string StateOf(Device hw) {
  return hw.HeaterState() ? "H" : "h"
       + hw.BlowerState() ? "B" : "b"
       + hw.CoolerState() ? "C" : "c"
       + hw.HiTempAlarm() ? "H" : "h"
       + hw.LoTempAlarm() ? "L" : "l";
}
```

En utilisant cette fonction, nous pouvons réduire le test à une seule assertion. Dès lors que vous connaissez la signification des lettres, votre regard glisse sur la chaîne et vous interprétez rapidement les résultats.

```
void TestTooCold()
{
    // arranger
    Device hw = new Device();
    hw.SetTemp(WAY_TOO_COLD);

    // agir
    hw.Tic();

    // auditer
    Assert.AreEqual( "HBchL", StateOf(hw) );
}
```

La construction du langage de test ne doit pas se limiter aux assertions. Au lieu de se fonder sur les API employées par les programmeurs pour manipuler le système, nous construisons un ensemble de fonctions et d'outils qui exploitent ces API. Cet ensemble devient une API spécialisée que les programmeurs utilisent pour simplifier l'écriture et la lecture des tests. Cette API de test n'est pas conçue à l'avance. Elle évolue à partir du remaniement continu du code de test, lorsque il est trop encombré de détails. Les développeurs disciplinés remanieront régulièrement leur tests afin d'en obtenir une forme plus succincte et expressive.

Exercice 2.3

La « chasse au Wumpus » est un jeu d'aventure en mode texte, consistant en une partie de cache-cache avec un monstre, le Wumpus, tapi dans un réseau de cavernes souterraines. A chaque tour, le joueur peut se déplacer dans une caverne adjacente ou tirer une flèche. Par déduction, le joueur doit localiser la salle dans laquelle se trouve le Wumpus, sans y avoir jamais pénétré. À ce point, il lance une flèche dans la direction supposée du monstre. Si le Wumpus s'y trouve, il remporte la partie ; sinon, la partie est perdue. L'objectif de cet exercice est de développer le jeu « chasse au Wumpus » selon une approche incrémentale pilotée par les tests unitaires.

1. Le terrain de jeu consiste en un quadrillage de salles, chacune étant reliée à quatre autres. Implémentez cela en utilisant ce test : « *étant donné* une grille de 5×5 salles et le joueur est dans la salle 1, *lorsque* on déplace le joueur vers l'est, *alors* il doit se trouver dans la salle 2 ». Voici le code de test correspondant, où les salles sont numérotées de gauche à droite et du haut en bas, à partir de l'indice zéro.

```
[TestClass]
public class GameTest {
   Game g;

   [TestInitialize]
   public void SetUp() {
      g = new Game(5, 5);
   }
   [TestMethod]
   public void GoEast() {
      g.PutPlayerInRoom(1);                    // étant donné...
      g.Move(Direction.EAST);                  // lorsque...
      Assert.AreEqual(2, g.GetPlayerRoom()); // alors...
   }
}
```

Testez aussi d'autres déplacements en horizontal et vertical.

```
public void GoWestTwice() {
   g.PutPlayerInRoom(1);
   g.Move(Direction.WEST);
   g.Move(Direction.WEST);
   Assert.AreEqual(4, g.GetPlayerRoom());
}
```

D'or en avant, tous les tests serons réalisés sur une grille de 5×5 salles.

2. Les cavernes présentent quelques pièges : les *puits sans fond*. Le joueur peut sentir la présence d'une piège si celle-ci se trouve dans l'une des salles adjacentes. Pour coder cela, utilisez les tests suivants.
   — Test `HearPit` : *étant donné* le jouer est dans 1 et un puits est dans 3, *lorsque* on déplace le joueur vers l'est, *alors* il doit sentir le puits ; puis, *lorsque* on le déplace vers l'ouest, il ne la sent plus.
   — Test `FallInPit` : *étant donné* le jouer dans 1 et un puits dans 2, *lorsque* on déplace le joueur vers l'est, *alors* il trouve le puits.
3. Le joueur peut sentir la présence du Wumpus s'il se trouve dans l'une des salles adjacentes. Pour coder cela, utilisez les tests suivants.
   — Test `SmellWumpus` : *étant donné* le jouer est dans 1 et le Wumpus est dans 3, *lorsque* le joueur bouge vers l'est, *alors* il doit sentir le Wumpus ; puis, *lorsque* il bouge vers l'ouest, il ne le sent plus.
   — Test `KilledByWumpus` : *étant donné* le joueur est dans la salle 1 et le Wumpus est dans la salle 2, *lorsque* on déplace le joueur vers l'est, *alors* le Wumpus est trouvé.
4. Le joueur peut tirer une flèche dans l'une des salles adjacentes. Si le Wumpus n'est pas dans la salle cible, la flèche continue et traverse 4 salles. La partie est gagnée si la flèche passe par la caverne du Wumpus.
   — Test `ShootArrow` : *étant donné* le joueur est dans la salle 1 et le Wumpus est dans la salle 2, *lorsque* on tire une flèche vers le sud, *alors* elle arrive dans la salle 21.
   — Test `KillWumpus` : *étant donné* joueur $= 1$ et Wumpus $= 2$, *lorsque* on tire une flèche vers l'est, *alors* le Wumpus est frappé.

2.1.3 Doublures de test

L'acte d'écrire les tests en premier permet souvent de découvrir des parties du système qui doivent être découplées. Par exemple, dans le jeu « Chasse au Wumpus », les cavernes présentent un deuxième type de pièges, les *chauves-souris*, qui transportent le joueur dans une autre salle sélectionnée aléatoirement. En écrivant d'abord le test correspondant, nous pouvons constater la difficulté de vérifier dans quelle salle le joueur a été transporté.

```
[TestMethod]
public void BatsCarryYouAway()
{
    g.PutPlayerInRoom(1);
    g.PutBatsInRoom(2);

    g.Move(Direction.EAST);

    Assert.AreEqual( ???????, g.GetPlayerRoom() ); // Où est-il ?
    Assert.IsTrue(g.bats_found);
}
```

Le problème est que le test se focalise sur deux concepts : (1) les chauves-souris transportent le joueur dans une autre salle ; (2) la nouvelle salle est choisie aléatoirement. Pour faire en sorte que le test ne sollicite qu'un seul composant du système à la fois, nous devons découpler ces deux concepts. Pour mieux comprendre ce propos, considérons cette implémentation naïve.

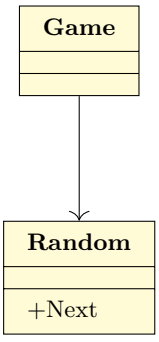
```
public class Game
{
    Random rng = new Random();
    ...

    public void Move(Direction d) {
        player.Move(d);
        PlayTurn();
        arrow_thrown = false;
    }

    void PlayTurn() {
        CheckForBats();
        CheckForWumpus();
        CheckForPits();
    }

    void CheckForBats() {
        bats_found = false;
        if (bats.Contains(player.Location)) {
            TransportPlayer();
            PlayTurn();
            bats_found = true;
        }
    }

    void TransportPlayer() {
        int room = rng.Next(width*height);
        PutPlayerInRoom(room);
    }
}
```

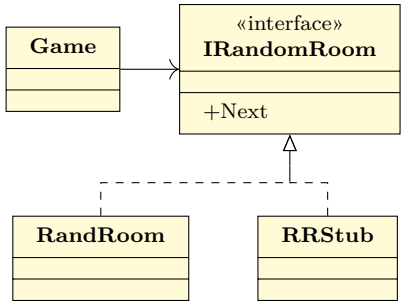


Dans l'implémentation ci-dessus, la classe `Game` présent une dépendance directe vers la classe `Random`, ce qui empêche de tester en isolation le comportement attendu (les chauves-souris transportent le joueur dans une autre salle). La solution consiste alors à introduire une « couche d'abstraction » en remplaçant le type `Random` par l'interface `IRandomRoom`, dont l'objet concret de type approprié sera injecté de l'extérieur par un setteur.

```
public class Game
{
    IRandomRoom rng;
    ...

    public void SetRR(IRandomRoom rng){
        this.rng = rng;
    }

    void TransportPlayer() {
        int room = rng.Next();
        PutPlayerInRoom(room);
    }
}
```



Avec cette nouvelle implémentation, nous pouvons facilement écrire le test par le biais d'un objet factice de type `RRStub:IRandomRoom` ayant un comportement contrôlé. Nous testons ainsi la classe `Game` en isolation.

```
[TestMethod]
public void BatsCarryYouAway()
{
  g.PutPlayerInRoom(1);
  g.PutBatsInRoom(2);
  g.SetRR( new RRStub(5) );

  g.Move(Direction.EAST);

  Assert.IsTrue(g.bats_found);
  Assert.AreEqual(5,g.GetPlayerRoom());
}
```

```
class RRStub : IRandomRoom
{
    public int room;

    public RRStub(int r) {
        this.room = r;
    }

    public int Next() {
        return room;
    }
}
```

De manière générale, quand la classe à tester dépend d'autres composants, nous devons simuler ceux-ci au travers de doublures. Une *doublure* est un objet factice utilisé dans un test unitaire pour obtenir un comportement reproductible, cela afin de garantir l'isolation du composant testé et l'indépendance entre les différents tests. Il existe deux types de doublures.

1. Le **bouchon** (« *stub* ») est une nouvelle classe écrite à la main qui tient compte du contexte exact pour lequel on a besoin d'une doublure. Tout le système est considéré comme une boîte blanche (on sait comment il est implémenté), afin d'écrire le bouchon le plus simple possible, avec le minimum de code nécessaire pour qu'il puisse remplir son rôle.
2. Le **Simulacre** (« *mock* ») est une classe qui définit les différents réactions attendues quand elle est manipulée par la classe testée. A différence des bouchons, un simulacre redéfinit le comportement d'une classe existante, afin de pouvoir contrôler les conditions de test.

Une doublure peut faciliter l'écriture de tests dans plusieurs situations. C'est notamment le cas lorsque la classe testée fait appel à un composant coûteux à mettre en place ou qui n'est lui-même pas encore développé, lorsque le test vise à vérifier le comportement d'une classe dans une situation exceptionnelle, ou lorsque la classe testée fait appel à du code non-déterministe.

En conclusion, les doublures favorisent un meilleur découplage entre les classes et augmentent la qualité des tests. Toutefois, elles nécessitent de suivre les changements dans les classes originales. De plus, lorsqu'une classe fonctionne avec une doublure, cela ne garantit pas que la classe fonctionnera quand il s'agira d'interagir avec la véritable implémentation. Pour vérifier cette seconde propriété, il faudra recourir à des *tests d'intégration*.

Exercice 2.4

Continuez l'exploration du jeu « Chasse au Wumpus » en ajoutant des nouveaux tests. Pour les réussir, il est nécessaire l'utilisation de bouchons afin de remplacer l'aspect aléatoire du jeu avec un comportement reproductible.

1. Les cavernes présentent un deuxième type de pièges : les chauves-souris. Pour coder cela, utilisez les tests suivants.
   - Test `HearBats` : *étant donné* le jouer est dans la salle 1 et un chauve-souris est dans la salle 3, *lorsque* on déplace le joueur vers l'est, *alors* il doit sentir le chauffe-souris ; puis, *lorsque* on le déplace vers l'ouest, il ne le sent plus.
   - Test `BatsCarryYouAway` : *étant donné* le jouer est dans 1, un chauve-souris est dans 2 et la salle sélectionnée aléatoirement est toujours la numéro 5, *lorsque* on déplace le joueur vers l'est, *alors* il doit trouver le chauve-souris et être transporté dans la salle 5.
2. Le joueur peut tirer une flèche dans l'une des salles adjacentes. Si le Wumpus n'est pas dans la salle cible, la flèche continue jusqu'à traverser 4 salles **sélectionnées aléatoirement**. Si la flèche ne passe pas par la caverne du Wumpus, ceci se déplace dans une autre salle choisie au hasard. Pour coder cela, utilisez les tests suivants.
   - Test `ShootArrow` : *étant donné* le joueur est dans 1, le Wumpus est dans 2 et la direction choisie au hasard est toujours l'est, *lorsque* on tire une flèche vers le sud, *alors* elle arrive à la salle 9.
   - Test `KilledByArrow` : *étant donné* le joueur est dans 1, le Wumpus est dans 2 et la direction choisie au hasard est toujours l'est, *lorsque* on tire une flèche vers l'ouest, *alors* le jouer doit être frappé.
   - Test `MoveWumpus` : *étant donné* le joueur est dans la salle 1, le Wumpus est dans la salle 2 et la direction choisie au hasard est toujours l'est, *lorsque* on tire une flèche vers le sud, *alors* le Wumpus n'est pas frappé et il doit se déplacer dans la salle 3.

Pour réaliser ces tests, il est nécessaire de créer un autre bouchon qui contrôle la direction de la flèche.

2.1.4 Tests de bout en bout

Dans sa forme la plus pure, le processus TDD commence par la spécification du logiciel en termes de phrases simples permettant de décrire avec suffisamment de précision le contenu des fonctionnalités souhaitées. Puis, une fonctionnalité est sélectionnée pour le développement, entrainant la construction progressive d'une suite spécifique de tests, afin d'accueillir l'implémentation correspondante dans le système. Ce cycle continue jusqu'à ce que toutes les fonctionnalités soient testées et implémentées.

Les tests unitaires nous permettent de vérifier le fonctionnement de chaque morceaux de code. Ils s'enfoncent dans les entrailles du système en appelant des méthodes spécifiques de classes particulières. Cependant, ils ne nous donnent pas une confidence complète que le système marche en totalité. Pour assurer cela, il faudra recourir à une granularité plus élevée que les tests unitaires, afin de solliciter du plus petit objet au système tout entier.

Les *tests de bout en bout* assurent que les composants du logiciel fonctionnent ensemble tel que prévu. Ils invoquent le système de plus loin, au niveau de l'API ou encore de l'IHM. L'application complète est testée par un scénario d'utilisation complet allant d'un bout à l'autre du système. Une bonne suite de tests est donc construite hiérarchiquement :

1. les **tests de bout en bout** vérifient que le système marche dans sa totalité (y compris son déploiement) ;
2. les **tests d'intégration** assurent que les composants du système interagissent correctement entre eux ;
3. les **tests unitaires** contrôlent que les classes logicielles du système soient correctement implémentées.

Exercice 2.5

Ajoutez au jeu « Chasse au Wumpus » des tests de bout en bout concernant : (1) plusieurs déroulements complets du jeu, avec le joueur gagnant ou perdant ; (2) une interface à ligne de commande. Voici un exemple du jeu.

```
You are in room 1.
Exits go to: 2 [East], 0 [West], 6 [South], 21 [North].
Do you want to [Move] or [Shoot] ? Move
Where ? East
------------------------------------------------------
You are in room 2.
You feel a cold wind blowing from a nearby cavern.
Exits go to: 3 [East], 1 [West], 7 [South], 22 [North].
Do you want to [Move] or [Shoot] ? Move
Where ? South
------------------------------------------------------
You are in room 7.
You smell something terrible nearby.
Exits go to: 8 [East], 6 [West], 12 [South], 2 [North].
Do you want to [Move] or [Shoot] ? Shoot
Where ? East
------------------------------------------------------
YOU KILLED THE WUMPUS! GOOD JOB, BUDDY!!!
```

2.2 Mini-projet : tetris

L'objectif de cette étude est de développer un jeu de tetris. Le but du jeu est de réaliser des lignes complètes en déplaçant des pièces de formes différentes, les *tétriminos*, qui défilent de haut en bas sur l'écran. Les lignes complétées disparaissent et le joueur peut de nouveau remplir les cases libérées. Le jeu n'a pas de fin : il faut résister le plus longtemps à la chute continue des tétriminos, jusqu'à ce que l'un d'entre eux reste bloqué en haut.

INSTRUCTIONS Pour réaliser le jeu de tetris, nous vous fournissons une solution Visual Studio organisée en deux dossiers. Le dossier « Test » contient des tests unitaires qu'il faut décommenter l'un après l'autre afin d'implémenter le jeu dans le dossier « Source ». Le but de travailler avec des tests pré-écrits est de s'habituer au cycle TDD, ainsi que de mieux comprendre le type de tests qu'il vous sera demandé d'écrire vers la fin de cette étude.

PREMIER TEST A l'ouverture de la solution Visual Studio, lorsque vous exécutez les tests pour la première fois, vous devez voir le test *board_is_empty* échouer. Corrigez le code de la classe `Board` (un changement d'une ligne) et ré-exécutez le test pour vérifier qu'il réussit.

2.2.1 Chute de blocs simples

La première étape consiste à implémenter la mécanique fondamentale du tetris : la chute d'un bloc. Les tests dans la classe `Step1_FallingBlocksTest` se limitent à la gestion d'un bloc simple de taille 1x1. Les blocs plus complexes seront traités dans une étape ultérieure. A cet égard, il est préférable d'éviter la création de nouvelles classes s'il n'est pas nécessaire. En fait, pour réussir les tests de cette première étape, il suffit d'implémenter deux classes : `Board` et `Block`. Plus tard, quand un vrai besoin de nouvelles classes se présentera, il sera possible de remanier le code afin d'aboutir à une meilleure conception.

2.2.2 Tétriminos

La deuxième étape consiste à définir les *tétriminos* et leurs rotations. En considérant les rotations comme équivalentes, il y a sept formes possibles :

- **Forme I** : ligne de quatre carrés alignés.
- **Forme T** : trois carrés en ligne, avec un quatrième carré sous le centre.
- **Forme L** : trois carrés en ligne, avec un quatrième carré sous le côté gauche.
- **Forme J** : trois carrés en ligne, avec un quatrième carré sous le côté droit.

— **Forme S** : deux lignes de deux carrés, dont la rangée supérieure est glissée d'un pas vers la droite.

— **Forme Z** : deux lignes de deux carrés, dont la rangée supérieure est glissée d'un pas vers la gauche.

— **Forme O** : Méta-carré de 2x2.

Les tétriminos peuvent avoir 1, 2 ou 4 orientations différents (voir la figure).

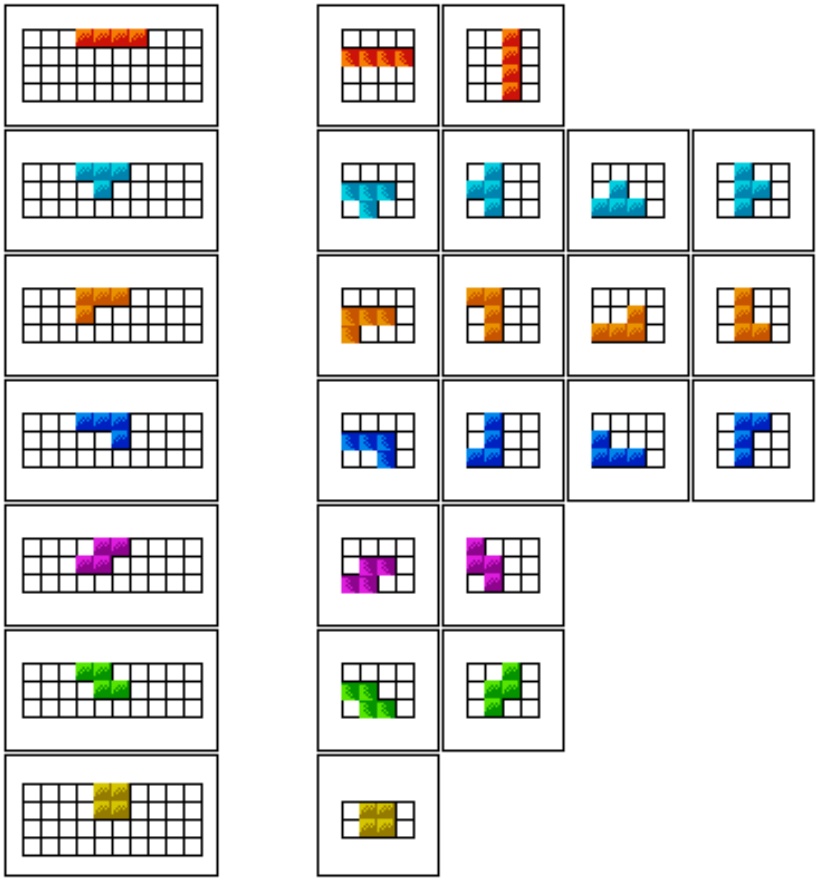


Figure 2.1 *Les 7 tétriminos, avec leurs positions initiales et leur orientations possibles.*

La classe `Step2_TetrominoesTest` fournit les tests pour le tétrimino T. Implémenter la classe `Tetromino` afin de réussir ces tests. Ensuite, créer un objet immuable pour chacun des 7 tétriminos. Suivre **exactement** les motifs dans la figure : par exemple, le tétrimino S doit être représenté par deux chaines de caractères : `"....\n.SS.\nSS..\n"` et `"S...\nSS..\n.S..\n"`. Enfin, remaniez le code de production et de test, afin d'éviter les duplications.

2.2.3 Chute de tétriminos

La troisième étape consiste à intégrer les tétriminos dans la mécanique de « chute d'un bloc » précédemment implémentée. On remarquera que, pour pouvoir réussir le premier test dans `Step3_FallingTetrominsTest`, il est nécessaire de remanier le code existant. Voici comment procéder, en veillant à ce que tous les tests réussissent après chaque petit changement.

1. Implémenter une classe `Piece` qui représente un tableau bidimensionnel (matrice) de caractères et qui réalise l'interface `Grid` suivante.

```
public interface Grid {
   int  Rows();
   int  Columns();
   char CellAt(int row, int col);
}
```

   La classe `Piece` doit avoir un constructeur qui convertit une chaine de caractères du type `"...\n...\n...\n"` dans une matrice. Elle doit également redéfinir la méthode `ToString()` qui reconvertit la matrice dans une chaine de caractères du type `"...\n...\n...\n"`. Pour implémenter ces méthodes, écrire d'abord les tests correspondants.
2. Remanier la classe `Tetromino` afin que : (i) elle stocke les tétriminos en utilisant des objets `Piece` ; (ii) elle réalise l'interface `Grid`.
3. Remanier la classe `Block` de manière qu'elle réalise l'interface `Grid`.
4. Dans la classe `Board`, changer la signature de la méthode `Drop` en `Drop(Grid shape)`. Afin de pouvoir compiler et réussir les tests des étapes précédentes, créer un objet `Block` au début de la méthode `Drop` en transmutant l'objet `Grid` passé en paramètre.
5. Dans la classe `Step1_FallingBlocksTest`, utiliser `Tetromino` à la place de `Block`, ce qui implique de remplacer la commande `new Block(c)` par `new Tetromino(c+"\n")` dans la fonction `DropBlock`.
6. Renommer la classe `Block` en `MovableGrid`. Étendre la gestion des blocs à une taille quelconque NxM, et inclure ici la logique de déplacement des tétriminos (attributs pour stocker le tetromino actif et sa position courante, fonctions pour vérifier les collisions avec les bords et les blocs déjà placés, etc). Remanier la classe `Board` en conséquence.
7. Décommenter le premier test dans `Step3_FallingTetrominsTest` et implémenter le code pour le réussir. L'une des difficultés est d'aligner le tétrimino à la première ligne de la planche. Voici une petite suggestion.

```
static int StartingRowOffset(Grid shape) {
   for (int r = 0; r < shape.Rows(); r++) {
      for (int c = 0; c < shape.Columns(); c++) {
         if (shape.CellAt(r, c) != '.') return -r;
      }
   } return 0;
}
```

8. Procéder aux tests suivants.

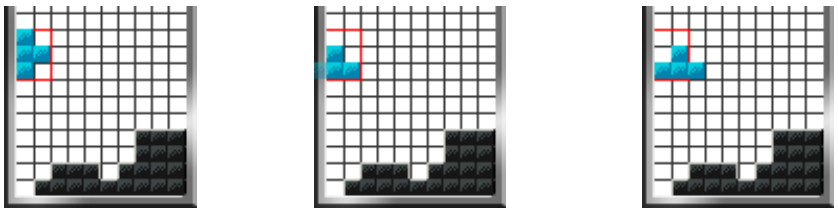
2.2.4 Déplacement du tétrimino pendant la chute

La quatrième étape consiste à implémenter la possibilité de déplacer à droite ou gauche le tétrimino qui est en train de chuter. Les tests pour réaliser cette tache sont fournis dans la classe `Step4_MoveFallingTetrominoesTest`. On remarquera que certains tests sont déclarés, mais ils ne sont pas écrits. C'est bien à vous de rédiger ceux-ci avant de passer à l'implantation du code.

2.2.5 Rotation du tétrimino pendant la chute

La cinquième étape consiste à implémenter la possibilité de pivoter le tétrimino qui est en train de chuter. Les tests pour réaliser cette tache sont fournis dans la classe `Step5_MoveFallingTetrominoesTest` (ils sont tous déjà écrits cette fois). A cet égard, il est conseillé de changer la signature de la méthode `Board.Drop` en `Drop(Tetromino shape)`.

On remarquera que les derniers test demandent de gérer les rotations à l'aide d'une technique nommée *wallkick*. Elle consiste à effectuer un déplacement juste après une rotation, afin d'éviter les obstacles pouvant empêcher le mouvement rotatif. La figure suivante illustre la rotation avec *wallkick* d'un tétromino T qui est adjacent au bord gauche.



(a) Situation de départ **(b)** Rotation sans *wallkick* **(c)** Rotation avec *wallkick*

Le code ci-dessous montre comment gérer le *wallkick* : une fois pivoté le tétrimino, il suffit de prendre en compte l'un des 5 déplacements possibles.

```
void TryRotate(MovableGrid rotated)
{
    MovableGrid[] moves = {
        rotated,
        rotated.MoveLeft(),  // wallkick moves
        rotated.MoveRight(),
        rotated.MoveLeft().MoveLeft(),
        rotated.MoveRight().MoveRight(),
    };
    foreach (MovableGrid test in moves)
    {
        if (!ConflictsWithBoard(test)) {
            fallingBlock = test;
            return;
        }
    }
}
```

2.2.6 Suppression de lignes complétées

Avant de passer à la sixième étape, il convient de tester le code dans sa globalité. Pour ce faire, il faut : (i) decommenter le code dans la classe `Program` ; (ii) compiler le projet « source » ; (iii) exécuter le jeu, dont le graphisme en ligne de commande est un hommage à la première version du tetris crée par *Alekseï Pajitnov*. A l'exécution, on remarquera que les lignés complétée ne sont pas supprimées ! Pour implémenter cela, il faut d'abord rédiger des tests (un à la fois) couvrant les différents cas de figure. Voici un exemple concernant la suppression d'une seule ligne.

```
[TestClass]
public class Step6_RemovingRowsTest
{
    Board board;
    Tetromino piece;

    [TestInitialize]
    public void SetUp()
    {
        board = new Board(6, 8);
        piece = new Tetromino(
            ".X.\n" +
            ".X.\n" +
            ".X.\n"
        );
    }

    void DropAndPushDown()
    {
        board.Drop(piece);
        while (board.IsFallingBlock())
            board.Tick();
    }

    [TestMethod]
    public void one_row_is_removed_and_the_empty_space_is_filled()
    {
        board.FromString(
            "........\n" +
            "........\n" +
            "........\n" +
            "AA.A.AAA\n" +
            "BBBB.BBB\n" +
            "CCCC.C.C\n"
        );

        DropAndPushDown();

        Assert.AreEqual(board.ToString(),
            "........\n" +
            "........\n" +
            "........\n" +
            "........\n" +
            "AA.AXAAA\n" +
            "CCCCXC.C\n"
        );
    }
}
```

2.3 Conclusion

A la fin de cette étude, vous aurez atteint le niveaux de *débutant* en TDD, ce qui correspond à savoir écrire un test unitaire et le code pour le réussir. Cependant, vous aurez probablement encore des difficultés à travailler en cycles de courte durée, écrire des tests de facile compréhension, et garder le code propre. Voici quelques erreurs courantes de débutant : oublier de dérouler les tests fréquemment ; écrire de nombreux tests à la fois ; écrire des tests d'une granularité inadaptée ; écrire des tests non probants ; écrire des tests pour du code trivial (par exemple des accesseurs).

Une autre remarque est que la majorité des tests utilisés dans cette étude sont très semblables entre eux, en raison du fait qu'ils utilisent la méthode `ToString()` pour vérifier l'état interne du jeu. Il est donc conseillé de pratiquer le TDD dans d'autres contextes. Également, cette étude utilise une approche *bottom-up*, mais il existe aussi une démarche de type *top-down*, illustrée dans le livre *Growing Object-Oriented Software, Guided by Tests*.

La question fondamentale est alors : « comment s'améliorer ? » A ce propos, il existe plusieurs niveaux de performance individuels.

1. **Débutant**
   (a) Je sais écrire un test unitaire avant le code correspondant.
   (b) Je suis capable d'écrire le code permettant de faire passer un test.
2. **Intermédiaire**
   (a) Je pratique la correction de défauts pilotée par les tests : lorsqu'un défaut est détecté, j'écris le test le mettant en évidence avant de le corriger.
   (b) Je suis capable de décomposer une fonctionnalité à coder en un certain nombre de tests à écrire.
   (c) Je connais plusieurs «recettes» pour guider l'écriture de mes tests (par exemple, pour un algorithme récursif ou itératif, écrire d'abord le test pour le cas terminal).
   (d) je suis capable d'extraire des éléments réutilisables de mes tests unitaires afin d'obtenir un outil de test adapté à mon projet.
3. **Avancé**
   (a) Je suis capable d'élaborer une « feuille de route » pour une fonctionnalité complexe sous forme de tests envisagés, et de la remettre en question si nécessaire.
   (b) Je suis capable de piloter par les tests différents paradigmes de conception : objet, fonctionnel, par événements, etc.
   (c) Je suis capable de piloter par les tests différents types de domaines techniques : calcul, interface graphique, accès aux données, etc.