GRENOBLE INP UGA Ensimag

École nationale supérieure d'informatique et de mathématiques appliquées

Principes SOLID et tests unitaires

Mnacho Echenim

Objectif global

Avoir du code **robuste et maintenable**

- Robuste:
  - Le moins de bugs possible
  - S'il y a un bug, être capable de le détecter et corriger le plus rapidement possible
- Maintenable:
  - Simple à lire par une personne tierce
  - Pouvoir le modifier plus tard *en toute confiance*

GRENOBLE INP UGA Ensimag

Principes SOLID

Les principes SOLID

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov's Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- Une classe ne devrait être changée que pour une raison unique
  - *Il y a une unique spécification dont la modification cause la modification de la classe*

```
public class Circle
    {
        private double radius;
        public Circle() { ... }
        public int Area() { ... }
        public void Draw() { ... }
    }
```

- « C'est son diamètre qui caractérisera un cercle »
- « Il faudra mettre en paramètre la surface sur laquelle on veut dessiner le cercle »

```
public class Circle
    {
        private double diameter;
        public Circle() { ... }
        public int Area() { ... }
    }

public class CircleDrawer
    {
        private Circle circle;
        public void Draw(Surface surface) { ... }
    }
```

- Avantage: on sait clairement quelle classe changer quand une spécification change
- Attention: le nombre de classes dans un projet peut devenir très important

Open/closed principle

- Une classe doit être ouverte à l'extension et fermée à la modification
  - Une fois terminée l'implémentation d'une classe, on n'y touche plus
    - Sauf pour rendre le code plus efficace
    - Sauf pour corriger des bugs
- Avantages:
  - On n'introduit pas de nouveaux bugs une fois qu'une classe est terminée
  - On ne réécrit pas les tests unitaire d'une classe terminée

Un exemple

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

Source: http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle

Un exemple (suite)

```
public abstract class Shape
{
    public abstract double Area();
}
```

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

Liskov's substitution principle

- « Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S, where S is a subtype of T. »
  - *On doit pouvoir remplacer n'importe quelle classe dans du code par une classe dérivée, de façon transparente*
  - Avantage: il est garanti qu'une classe dérivée n'introduira pas de nouveau bug

Liskov’s substitution principle (suite)

- Conditions à vérifier
  - Les paramètres d’une méthode de la classe dérivée doivent être plus généraux que ceux de la classe parente
  - Le type de retour d’une méthode de la classe dérivée doit être plus spécifique que celui de la classe parente
  - Les exceptions de la classe dérivée doivent être dérivées de celles de la classe parente
  - Il ne faut pas ajouter des préconditions à une méthode de la classe dérivée
  - Il ne faut pas relâcher les postconditions d’une méthode de la classe dérivée
  - Il ne faut pas qu’une propriété *readonly* puisse être modifiée dans la classé dérivée, et vice versa

Le « contre-exemple »

- Une classe *Carré* qui hérite d'une classe *Rectange*
  - Un carré est un rectangle
  - Il faut implémenter *SetWidth* et *SetLength*
- Mais la longueur et la largeur ne sont plus indépendantes

```
Rectangle r = Container.GetObject();
r.SetWidth(5);
r.SetLength(10);
Console.WriteLine(r.Area()); // 50 or 100?
```

Interface Segregation Principle

- Une classe ne doit pas dépendre de méthodes dont elle n'a pas besoin
  - Ces méthodes sont définies dans des interfaces

```
public interface IOption
    {
        double GetPayoff(..);
        double GetPrice(..);
    }
    public class Call: IOption
    {
        public double GetPayoff(..) { .. }
        public double GetPrice(..) { .. }
    }
```

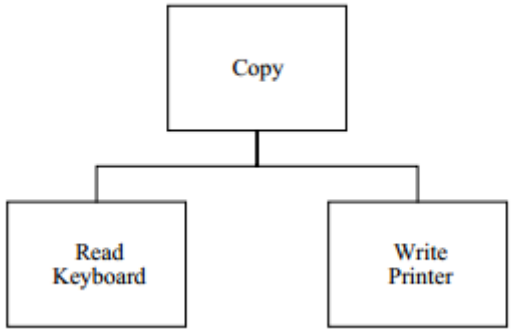
```
public interface IOption
    {
        double GetPayoff(..);
    }
    public interface IPriceable
    {
        double GetPrice(..);
    }
    public class Call: IOption, IPriceable
    {
        public double GetPayoff(..) { .. }
        public double GetPrice(..) { .. }
    }
```

Les interfaces sont plus lisibles et seules les méthodes nécessaires sont implémentées

Dependency Inversion Principle

- Les composants d'un programme doivent être reliés entre eux en passant par des abstractions (classes abstraites, interfaces)
  - Les abstractions décrivent le *quoi*, les implémentations décrivent le *comment*
- Illustration (*Clean Code* de Robert C. Martin)

Figure 1. Copy Program.



Dependency Inversion Principle

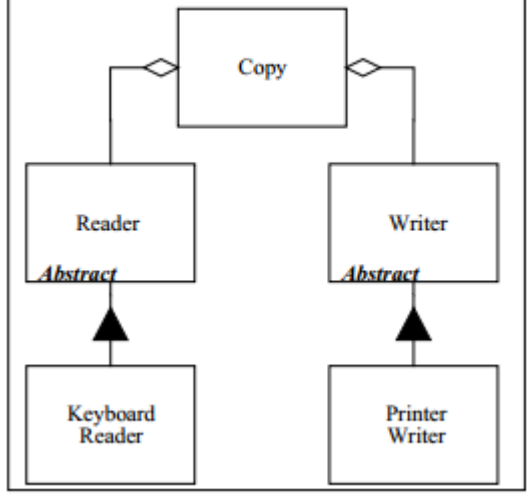
Listing 1. The Copy Program

```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

Listing 2. The "Enhanced" Copy Program

```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

Figure 2: The OO Copy Program



Listing 3: The OO Copy Program

```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

Recommandations

- Inversion de dépendance (*Inversion of control containers*)
  - MEF
  - Unity
- Lisibilité
  - Les noms des classes, variables, méthodes… doivent être explicites
  - Les corps des méthodes doivent être concis

Tests Unitaires

Une citation

« Les tests, ça ne rapporte pas beaucoup de points en projet donc on n'en fait pas beaucoup. Pourtant c'est super important dans le monde professionnel. »

Simon Ardiet, **Xrays Trading**, promo 2022

Tests unitaires

- *Documentation vivante*
  - Une base de tests se maintient
  - Les noms des tests doivent êtres explicites
- Test driven development
  - Implémentation des tests avant les classes
  - Attention: démarche non triviale!
- Behavior driven development
  - Une forme de TDD
  - Tests lisibles par tous les *stakeholders*
  - Ex: *Given-When-Then*

Que faut-il tester?

- *Pragmatic unit testing in C# with NUnit*: Right-BICEP
  - Are the results **Right**?
  - Are the **B**oundary conditions correct?
  - Is there an **I**nverse relationship that can be checked?
  - Can the results be **C**ross-checked?
  - Can **E**rror conditions be forced to happen?
  - Are the **P**erformance characteristics within bounds?

Conditions aux frontières

- *Boundary conditions should be CORRECT*
  - Conformance
    - Est-ce que les valeurs doivent respecter un certain format?
  - Ordering
    - L'ensemble des valeurs est-il ordonné? Est-ce important?
  - Range
    - Est-ce que la valeur est dans le bon intervalle min/max?
  - Reference
    - Est-ce que le code fait référence à des librairies externes?
  - Existence
    - Est-ce que la valeur existe (non-nulle, dans un ensemble…)?
  - Cardinality
    - Y a-t-il le bon nombre de valeurs?
  - Time
    - Est-ce que les étapes se déroulent dans le bon ordre? Au bon moment?

Vérification de relations inverses

- Utilisé typiquement pour tester des fonctions bijectives

```
public void TestSquareRoot
{
    double x = MyClass.MySquareRoot(4.0);
    Assert.That(4.0, Is.EqualTo(x*x).Within(1E-13));
}
```

- Attention aux mêmes bugs introduits dans la fonction et son inverse
  - Si possible, utiliser une autre implémentation pour la fonction inverse

Vérification croisée

- Utiliser d'autres moyens pour calculer la même valeur
- Autres sources:
  - Autres algorithmes
  - Librairies
  - Calculs à la main

Exemple

Pricer pour les options barrière

- Principe
  - Base: la même que pour le projet Couverture de Produits Dérivés
  - Cmake: compilation sour Windows et Linux
  - Tests unitaires: Google Test
  - Plusieurs étapes de refactoring
    - Ce n’est pas fini
  - Buggé?
    - Peut-être…
- Disponible sur Chamilo

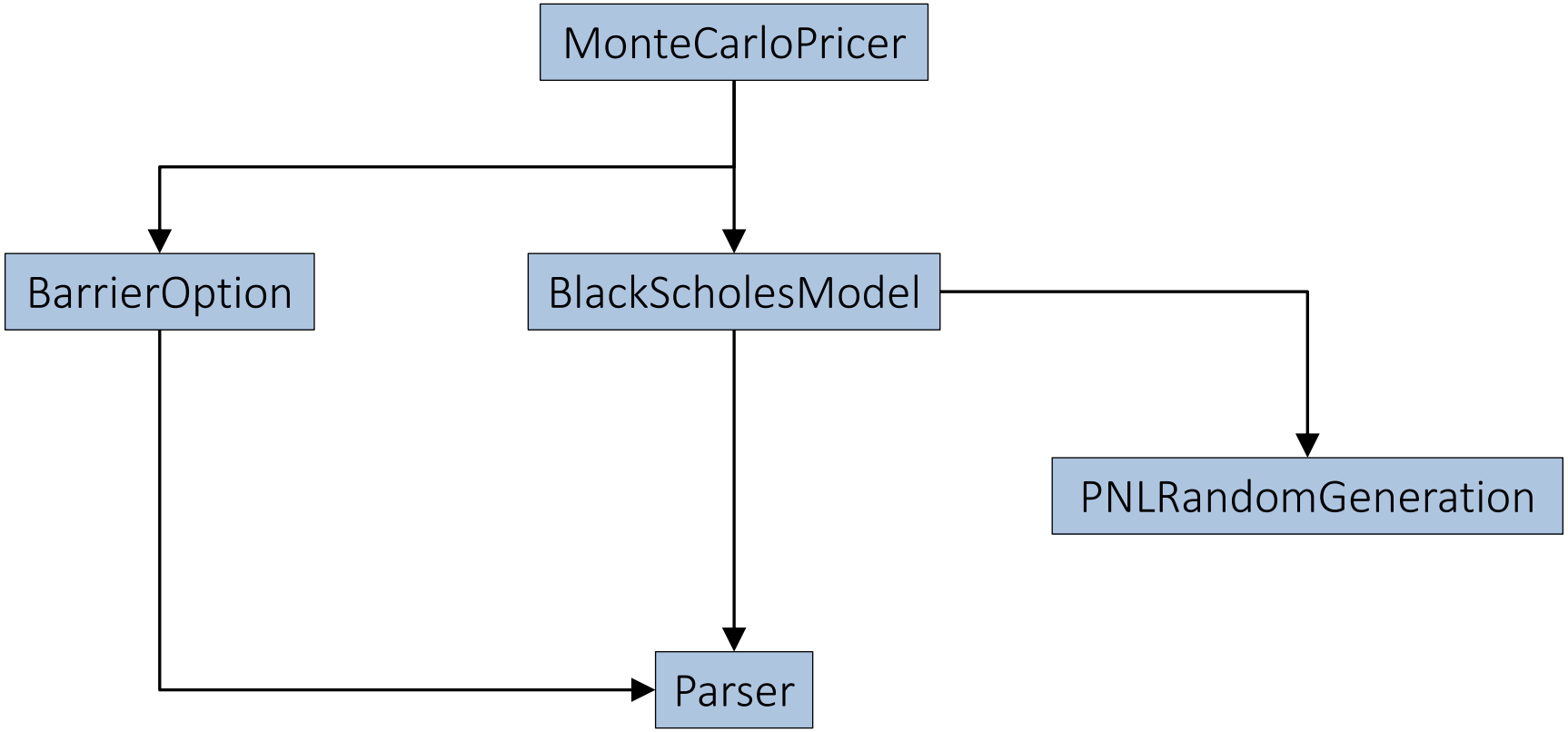
Objectif

- Code permettant de pricer une option barrière par la méthode de Monte Carlo:
  - En 0.
  - En un temps *t* quelconque.
- Paramètres de l'option récupérables depuis un fichier
- Paramètres du modèle de Black-Scholes pour les sous-jacents récupérables depuis un fichier
- Le code doit être testable/maintenable/extensible

Choix des classes

- MonteCarloPricer
  - Contient les deux méthodes principales: *price* (en 0) et *price_at* (en *t* quelconque)
- BarrierOption
  - Contient les caractéristiques de l'option et la fonction payoff
- BlackScholesModel
  - Contient les paramètres de simulation des sous-jacents
- Parser
  - Permet de lire les données dans un fichier
- RandomGeneration
  - Permet de simuler des variables aléatoires

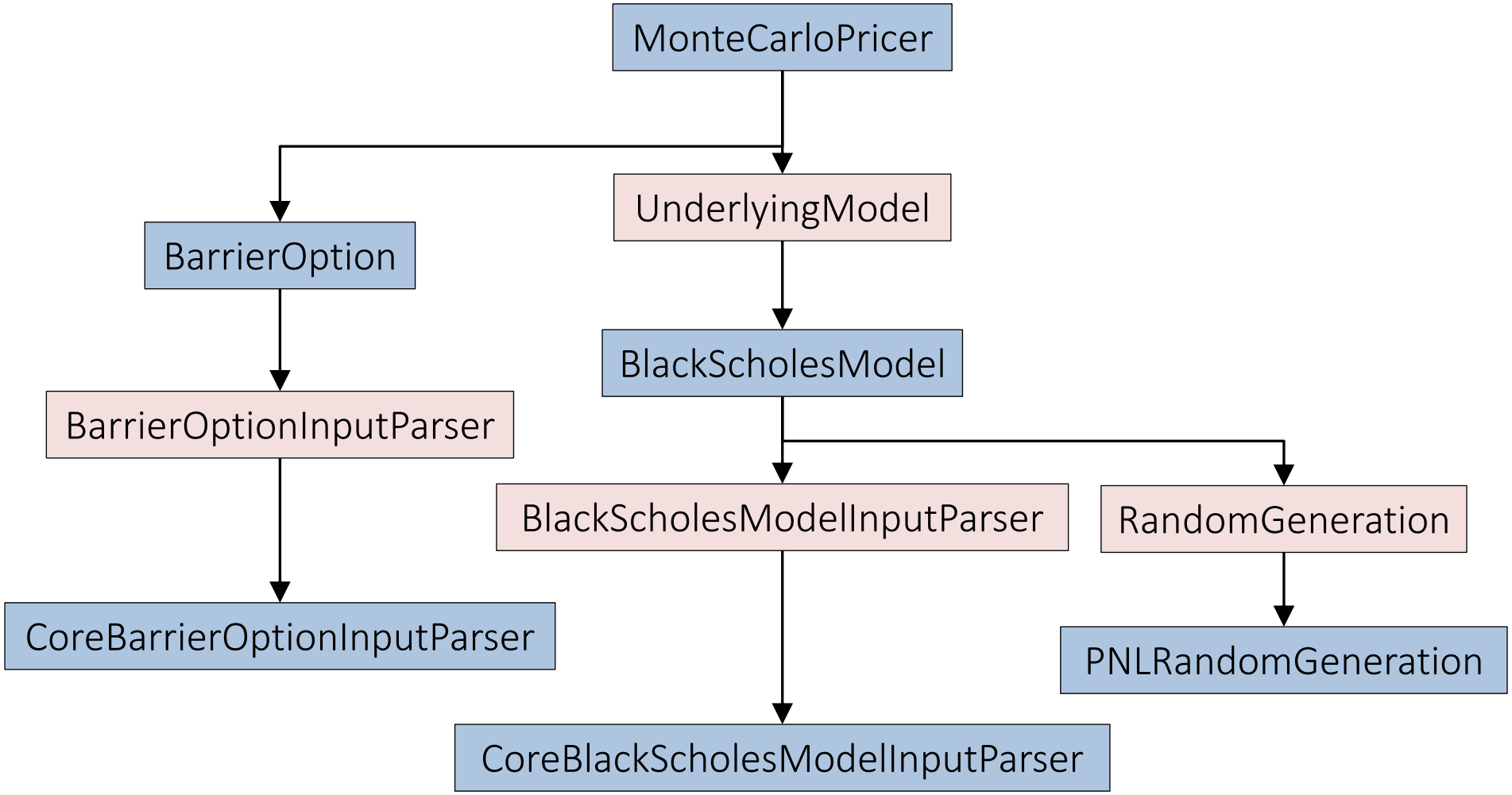
Dépendances



Modifications à apporter

- Single Responsibility Principle
  - Scinder le parseur en deux
- Dependency Inversion Principle: intercaler des classes abstraites
  - Code sera plus facile à étendre ensuite
  - Permet de faire des tests unitaires plus simplement (les dépendances sont réduites)
  - Exemples:
    - Instancier un objet BarrierOption pour le tester sans avoir besoin d'un fichier contenant les paramètres
    - Faire des simulations avec un faux générateur aléatoire

Dépendances: nouvelle version



Autres modifications

- BarrierOption:
  - Création d'une classe BarrierOptionParameters
  - BarrierOption contient un objet BarrierOptionParameters et une méthode *get_payoff()*
- BlackScholesModel:
  - Création d'une classe BlackScholesParametersHelper
    - Calcule la factorisée de Cholesky de la matrice de corrélation
    - Stocke les lignes de la factorisée pour une récupération plus efficace

Eviter la duplication de code

- Les méthodes de la classe MonteCarloPricer *price* et *price_at* ont la même structure
- Seule différence:
  - *price* prend en paramètre un vecteur de spots
  - *price_at* prend en paramètre une matrice de prix passés
- Deux patrons de conception possibles pour éviter la duplication de code:
  - Patron stratégie (strategy pattern)
  - Patron de méthode (template methode pattern)

Classe MonteCarloRoutine

- Contient le code commun aux deux méthodes de pricing

```
void MonteCarloRoutine::price(double &price, double &confidence_interval) const
{
      double runningSum = 0;
      double runningSquaredSum = 0;
      double payoff;
      for (unsigned long i = 0; i < sample_number; i++)
      {
            const PnlMat * const generated_path = get_generated_path();
            payoff = option.get_payoff(generated_path);
            runningSum += payoff;
            runningSquaredSum += payoff * payoff;
      }
      price = (…)
      confidence_interval = (…)
}
```

Méthode virtuelle

- Les classes dérivées de MonteCarloRoutine fournissent les implémentations spécifiques de *get_generated_path*

Classes dérivées

- MonteCarloRoutineAtOrigin

```
class MonteCarloRoutineAtOrigin : public MonteCarloRoutine
{
private:
	const PnlVect * const spots;
protected:
	const PnlMat * const get_generated_path() const
	{
		return underlying_model.simulate_asset_paths_from_start(spots);
	}
	(…)
};
```

- MonteCarloRoutineAtTimeT

```
class MonteCarloRoutineAtTimeT : public MonteCarloRoutine
{
private:
	const PnlMat * const past;
	const double t;
protected:
	const PnlMat * const get_generated_path() const
	{
		return underlying_model.simulate_asset_paths_unsafe(t, past);
	}
	(…)
};
```

Code factorisé

- Implémentation de MonteCarloPricer

```
void MonteCarloPricer::price(…) const
{
   MonteCarloRoutineAtOrigin mco(…);
   mco.price(price, confidence_interval);
}

void MonteCarloPricer::price_at(…) const
{
   MonteCarloRoutineAtTimeT mct(…);
   mct.price(price, confidence_interval);
}
```

Récapitulatif

- Ce qui peut être facilement étendu
  - Changer de générateur aléatoire
  - Récupération des paramètres dans d'autres formats, depuis d'autres sources
  - Utilisation de modèles de diffusion des sous-jacents autres que le modèle de Black-Scholes
- Etapes suivantes
  - Classe abstraite PathDependentOption
  - Abstraction des données du passé pour supprimer dépendance à PnLMat
  - Encapsulation option + modèle pour veiller plus simplement à la cohérence des paramètres du pricer
  - Encore des tests!
    - Pour le pricing en *t*

Dernières remarques

- N'implémenter que ce qui est nécessaire
  - Quitte à avoir plus de phases de refactoring
  - Ne rien anticiper *(« je ne sais plus pourquoi j'avais organisé mon code comme ça »)*
- Toujours écrire un test qui plante avant de le faire passer
  - Pour éviter de se retrouver avec des bugs pénibles
- Google Mock: pas si terrible quand j'ai testé