# TP - Pattern GRAPS

LE TP Fonctionne sous Windows, IOS et Linux, il vous suffit d’adapter les commandes, outils à chaque environnement. Toutes les étapes ne sont pas entièrement expliquées, le meilleur apprentissage est la recherche et le fait de faire des erreurs. Les moteurs de recherches sont vos meilleurs amis :

## Installation des outils

Sous Windows :

* dotnet core SDK dernière version
* Editeurs :
  + Visual Studio Entreprise 2017 (Version d’évaluation 1 mois)
    - Plugin visual studio Resharper (Version d’évaluation 1 mois, gratuit pour les étudiants)
  + Rider (Version d’évaluation 1 mois, gratuit pour les étudiants)
  + Visual Studio Code
* Git for windows

Sous Mac :

* dotnet core SDK dernière version
* Editeurs :
  + Visual Studio for mac
  + Rider (Version d’évaluation 1 mois, gratuit pour les étudiants)
  + Visual Studio Code
* Git

Sous Linux :

* Editeurs :
  + Visual Studio for mac
  + Rider (Version d’évaluation 1 mois, gratuit pour les étudiants)
* Git

**Le TP est réalisé pour que vous utilisiez principalement Visual Studio Entreprise 2017 ou Rider !**

## Création et récupération de votre git

Allez à l’adresse ci-dessous et suivez les instructions afin de créer votre espace git :  
<https://classroom.github.com/a/6piqMVVZ>

Vous obtiendrez une url pour votre nouvelle espace git.   
Si vous n’êtes pas inscrit sur github, inscrivez-vous.

Cloner votre espace git dans un répertoire projet et créer un répertoire « src » à la racine :

|  |
| --- |
| *cd ‘’C:/projects’’*  *git clone ‘’url-devotre-espace-git’’* *cd ‘’lenomdevotreespace’’* *mkdir ‘’src’’ //Créer le répertoire « src »* |

Ajouter un fichier « .gitignore » à la racine de votre projet qui permettra de pas sauvegarder les fichiers générés dynamiquement à la compilation (inutile à sauvegarder et cela pollue votre gestionnaire de source)

<https://www.google.fr/search?hl=fr&q=.gitignore+C%23>

N’oubliez pas de sauvegarder vos modifications dans git tout au long du TP. Votre évaluation sera réalisée par rapport aux sources présentes dans git.

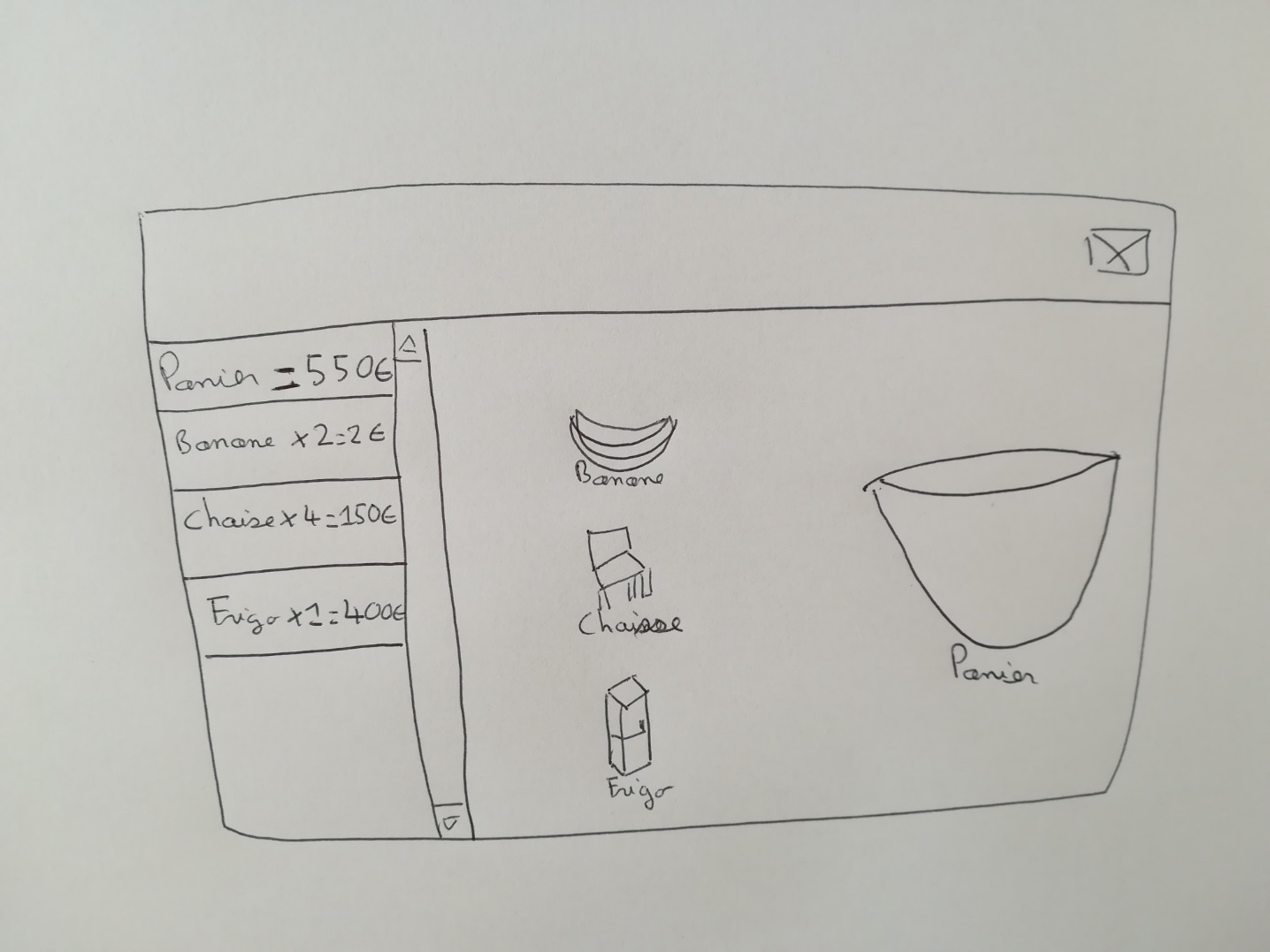
|  |
| --- |
| *cd ‘’C:/projects/lenomdevotreespace’’’* *git add . //* <https://git-scm.com/docs/git-add>  *git commit -m “mon message de commit” //* <https://git-scm.com/docs/git-commit>  *git push //* <https://git-scm.com/docs/git-push> |

## Début des spécifications techniques et fonctionnelles

Nous allons créer une application qui va permettre de mettre des articles dans un panier et de calculer le prix de ces articles. L’application fonctionnera sur desktop et sera compatible avec tous supports (IOS, Linux, Windows). L’objectif est de pouvoir « Drag&Dropper » avec la souris les articles dans le panier.

Les prix et stocks des articles sont fournis par une base de données. Pour le TP et par souci de facilité, la base de données sera un fichier au format JSON.

Le tout sera réalisé en TDD (Test Driven Development), c’est-à-dire que nous allons réaliser les tests avant le code.



L’utilisateur va pouvoir placer des articles dans un panier. Pour le moment (enfin pour ce TP) nous allons uniquement nous intéresser à l’opération qui permet de calculer le montant total du panier. « En fonction » des articles le prix total à payer va s’afficher. Il y a quelques règles à respecter :

* Un article de type “food”, possède une TVA à 12 %
* Un article de type “electronic” possède une TVA à 20 % et nécessite une éco-taxe de 4€ à l’achat
* Un article de type “desktop” possède une TVA à 20 %

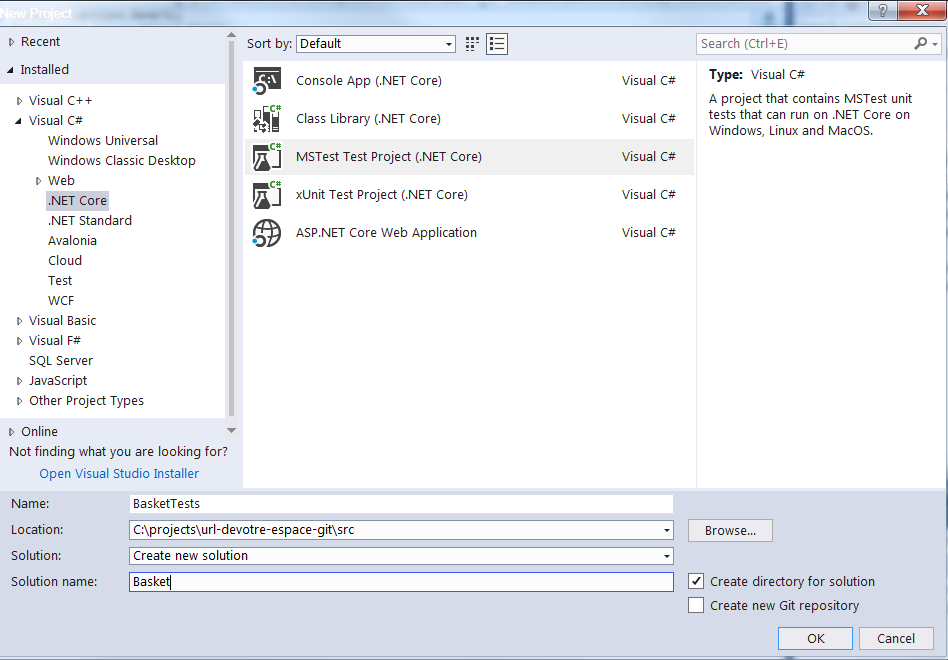
Exemples :

* Si l’on prend 12 bananes, 1 réfrigérateur et 4 chaises, le prix total est de 848,68€
* Si l’on prend 20 bananes et 6 chaises, le prix total est de 375,20€

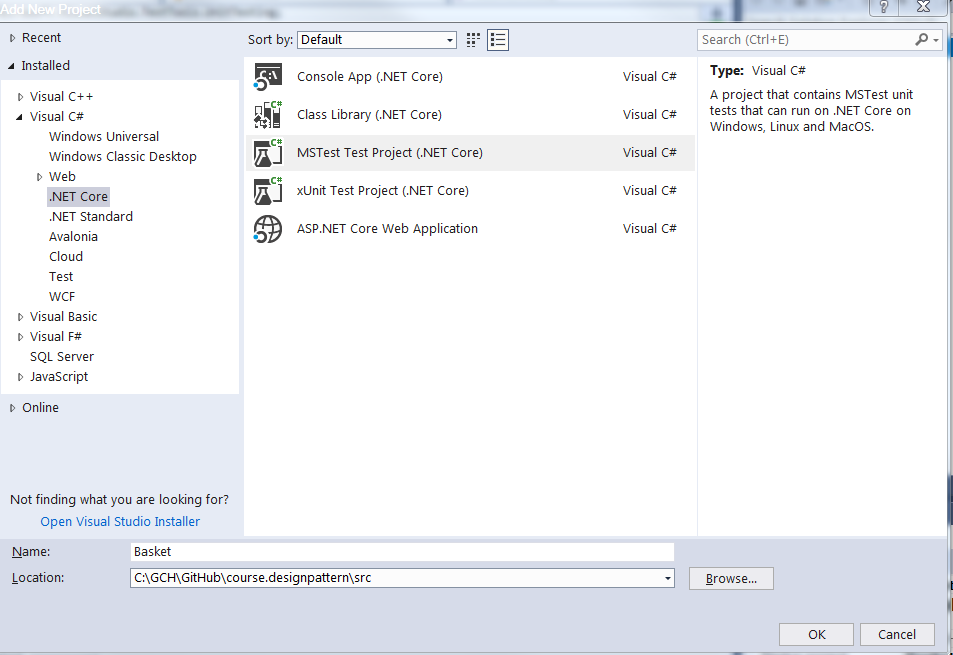
## Mise en place du projet

Pour démarrer : Créer un projet de .NET Core de type MSTest qui s’appellera « **BasketTest** », placer sa localisation dans « **C:\projects\url-devotre-espace-git\src** ». Le nom de la solution doit être « **Basket** ». La case « **Create a directory for solution** » doit être décochée.

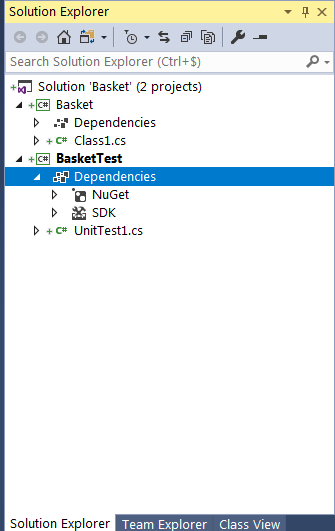
Cela va créer le fichier de projet « Basket.sln » et un répertoire projet de Test « BasketTest »



Ajouter maintenant à la solution une librairie de type « .NET Standard » Librairie, c’est là où vous placerez votre code. Nommer le « Basket ».



Vous avez maintenant la solution dans cet état :



Pour pouvoir tester la librairie « Basket », il faut ajouter une référence depuis « BasketTest » vers « Basket » (clic droit sur « **Dependencies** »).

Créer un fichier « ArticleDatabase.cs » à la racine du projet « Basket ». Cette structure renverra les données provenant de la base de données.

|  |
| --- |
| namespace Basket  {  public struct ArticleDatabase  {  public string Id { get; set; }  public string Label { get; set; }  public string Category { get; set; }  public int Price { get; set; }  public int Stock { get; set; }  }  } |

**Basket/ArticleDatabase.cs**

Créer un fichier « BasketLineArticle.cs » à la racine du projet « Basket ». Cette structure correspondra aux informations d’entrées envoyées qui vous permettront de savoir quels éléments vous avez dans votre panier.

|  |
| --- |
| namespace Basket  {  public struct BasketLineArticle  {  public string Id { get; set; }  public int Number { get; set; }  public string Label { get; set; }  }  } |

**Basket/BasketLineArticle.cs**

|  |
| --- |
| namespace BasketTest  {  [TestClass]  public class BasketOperation\_CalculateBasketAmoutShould  {  public class BasketTest  {  public List<BasketLineArticle> BasketLineArticles { get; set; }  public int ExpectedPrice { get; set; }  }  private static IEnumerable<object[]> Baskets  {  get  {  return new[]  { new object[] {  new BasketTest(){ BasketLineArticles = new List<BasketLineArticle>  {  new BasketLineArticle {Id = "1", Number = 12, Label = "Banana"},  new BasketLineArticle {Id = "2", Number = 1, Label = "Fridge electrolux"},  new BasketLineArticle {Id = "3", Number = 4, Label = "Chair"}  },  ExpectedPrice = 84868}  },  new object[] {  new BasketTest(){ BasketLineArticles = new List<BasketLineArticle>  {  new BasketLineArticle {Id = "1", Number = 20, Label = "Banana"},  new BasketLineArticle {Id = "3", Number = 6, Label = "Chair"}  },  ExpectedPrice = 37520}  },  };  }  }  [TestMethod]  [DynamicData("Baskets")]  public void ReturnCorrectAmoutGivenBasket(BasketTest basketTest)  {  var amountTotal = 0;  Assert.AreEqual(amountTotal, basketTest.ExpectedPrice);  }  }  } |

**BasketTest/BasketOperation\_CalculateBasketAmoutShould.cs**

Le test en l’état actuel doit échouer, car l’implémentation du code n’est pas présente.

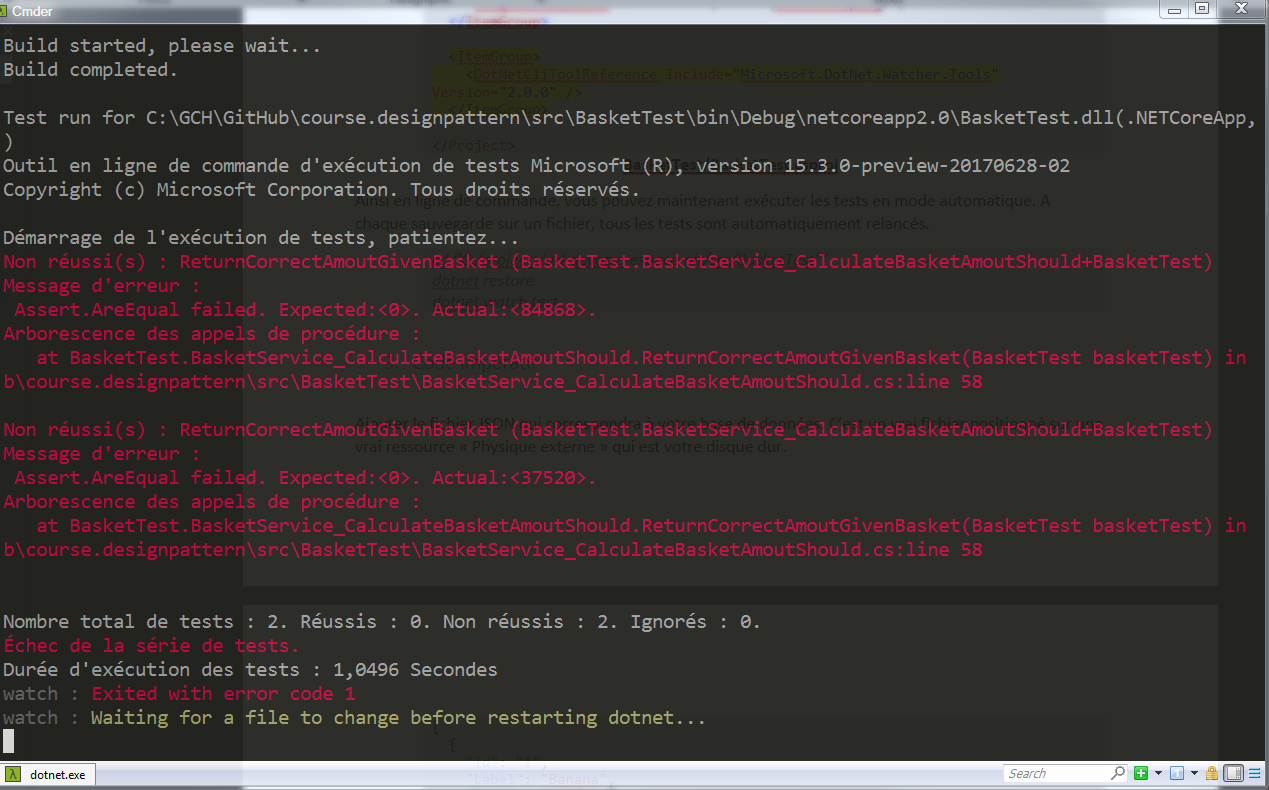
Si votre code ne compile pas, vérifier que les librairies utilisées sont bien à jour, comme ci-dessous. Ajouter l’outil qui va permettre de relancer les tests automatiquement (surligner en jaune).

|  |
| --- |
| <Project Sdk="Microsoft.NET.Sdk">  <PropertyGroup>  <TargetFramework>netcoreapp2.0</TargetFramework>  <IsPackable>false</IsPackable>  </PropertyGroup>  <ItemGroup>  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.6.0" />  <PackageReference Include="MSTest.TestAdapter" Version="1.2.0" />  <PackageReference Include="MSTest.TestFramework" Version="1.2.0" />  </ItemGroup>  <ItemGroup>  <ProjectReference Include="..\Basket\Basket.csproj" />  </ItemGroup>  <ItemGroup>  <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="2.0.0" />  </ItemGroup>  </Project> |

**BasketTest/BasketTest.csproj**

Ainsi en ligne de commande, vous pouvez maintenant exécuter les tests en mode automatique. A chaque sauvegarde sur un fichier, tous les tests sont automatiquement relancés.

|  |
| --- |
| *cd ‘’C:/projects/url-devotre-espace-git/src/BasketTest’’’*  *dotnet restore* *dotnet watch test* |



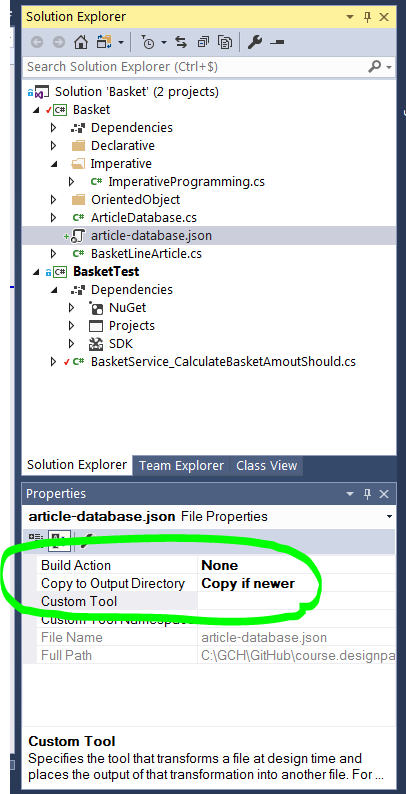
## Code impératif

Ajouter le fichier JSON qui correspondra à votre base de données. C’est un vrai fichier positionné sur une vraie ressource « Physique externe » qui est votre disque dur.

|  |
| --- |
| [  {  "Id": "1",  "Label": "Banana",  "Category": "food",  "Price": 1,  "Stock": 35  },  {  "Id": "2",  "Label": "Fridge electrolux",  "Category": "electronic",  "Price": 500,  "Stock": 20  },  {  "Id": "3",  "Label": "Chair",  "Category": "desktop",  "Price": 49,  "Stock": 68  }  ] |

**Basket/article-database.json**

Positioner l’attribut « Copy to output directory » à « Copy if newer ». Cela permettra à votre fichier d’être copié à la racine de votre répertoire d’exécution de votre application et/ou test à chaque compilation s’il a été modifié.



Mettez à jour le test avec la première implantation ci-dessous.

|  |
| --- |
| [TestMethod]  [DynamicData("Baskets")]  public void ReturnCorrectAmoutGivenBasket(BasketTest basketTest)  {  var amountTotal = 0;  foreach (var basketLineArticle in basketTest.BasketLineArticles)  {  // Retrive article from database  var codeBase = Assembly.GetExecutingAssembly().CodeBase;  var uri = new UriBuilder(codeBase);  var path = Uri.UnescapeDataString(uri.Path);  var assemblyDirectory = Path.GetDirectoryName(path);  var jsonPath = Path.Combine(assemblyDirectory, "article-database.json");  IList<ArticleDatabase> articleDatabases = JsonConvert.DeserializeObject<List<ArticleDatabase>>(File.ReadAllText(jsonPath));  var article = articleDatabases.First(articleDatabase => articleDatabase.Id == basketLineArticle.Id);  // Calculate amount  var amount = 0;  switch (article.Category)  {  case "food":  amount += article.Price \* 100 + article.Price \* 12;  break;  case "electronic":  amount += article.Price \* 100 + article.Price \* 20 + 4;  break;  case "desktop":  amount += article.Price \* 100 + article.Price \* 20;  break;  }  amountTotal += amount \* basketLineArticle.Number;  }  Assert.AreEqual(amountTotal, basketTest.ExpectedPrice);  } |

**BasketTest/BasketOperation\_CalculateBasketAmoutShould.cs**

Le test doit passer au « vert », cependant ce n’est pas un vrai test « unitaire », car il accède à une ressource physique qui est votre disque dur, lors de la lecture du fichier JSON.

L’algorithme utilise des entiers numériques car l’utilisation des nombres flottants entraine des erreurs de calculs car tronqués par les processeurs.

## Code impératif => Procédural & Maitriser son IDE

Vous ne devez pas toucher à votre souris pour réaliser cette partie. Prenez votre temps, l’objectif est que vous montiez en compétence et que vous appreniez à automatiser vos propres actions. 2h de montées en compétence pourront vous en faire gagner des milliers dans votre vie (et beaucoup sur ce TP) : <https://www.google.fr/webhp?q=resharper+keymap>

Le but est d’extraire le code du calcul au code de test. Faites cette extraction en utilisant au maximum les raccourcis clavier. Pour information ; CTRL+SHIFT+R est un raccourci qui vous donne accès à la liste complète des opérations disponibles, c’est le seul à connaître par cœur avec Resharper/Rider. Créer un fichier « ImperativeProgramming.cs » à la racine du projet « Basket ».

|  |
| --- |
| public class ImperativeProgramming  {  public static int CalculateBasketAmount(IList<BasketLineArticle> basketLineArticles)  {  // here your code implementation  }  } |

**Basket/ImperativeProgramming.cs**

Vérifier que votre test fonctionne toujours.

Le code d’appel au JSON est mélangé au reste de l’algorithme. Sortez le dans une autre méthode qui prend en paramètre l’identifiant de l’article.

|  |
| --- |
| public static ArticleDatabase GetArticleDatabase(string id)  {  // here your code implementation  } |
|  |

**Basket/ImperativeProgramming.cs**

Vérifier que votre test fonctionne toujours.

Il y a un problème, votre test n’est pas unitaire, il accède toujours à une ressource physique qui est le disque dur. Un test pour être unitaire ne doit pas avoir d’accès à de ressource externe (disque dur, appel sur le réseau, etc.). Il doit uniquement exécuter du code. Ajouter la méthode ci-dessous qui simulera l’accès à la base de données.

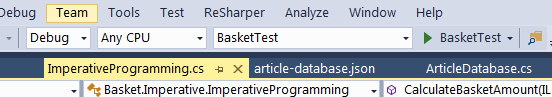
|  |
| --- |
| public static ArticleDatabase GetArticleDatabaseMock(string id)  {  switch (id)  {  case "1":  return new ArticleDatabase {Id = "1", Price = 1, Stock = 35, Label = "Banana", Category = "food"};  case "2":  return new ArticleDatabase  {  Id = "2",  Price = 500,  Stock = 20,  Label = "Fridge electrolux",  Category = "electronic"  };  case "3":  return new ArticleDatabase {Id = "3", Price = 49, Stock = 68, Label = "Chair", Category = "desktop"};  default:  throw new **NotImplementedException**();  }  } |

**Basket/ImperativeProgramming.cs**

Modifier le code qui appel la procédure pour ajouter une directive de compilation.

|  |
| --- |
| #if DEBUG  var article = GetArticleDatabaseMock(basketLineArticle.Id);  #else  var article = GetArticleDatabase(basketLineArticle.Id);  #endif |

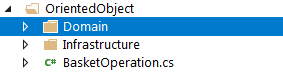
Maintenant, votre code est testable unitairement en mode « debug ». Par contre, cela engendre un nouveau problème : Votre code testé unitairement, est uniquement celui de la version « debug ». Votre code n’est pas le même en mode « debug » qu’en mode « release » qui sera livré à l’utilisateur.



## Code impératif => Code Orienté Objet

Le code est l’état actuel (Impératif+ procédurale) est peu cohérent. Il réalise beaucoup de choses. Il accède à la base de données et possède les règles de calcul du panier.

Nous allons maintenant mettre de l’orienté objet dans notre algorithme et augmenter la cohérence de chaque action. Notre code sera plus lisible, les règles métiers seront exprimées comme si c’était une documentation. On aimerait que chaque règle et responsabilités soient portées par les bons objets. On aimerait séparer la partie technique (base de données) de la partie « métier » des règles du panier (calcul).



Le répertoire « Domain » possèdera les objets en relation avec les règles de calcul. Il va contenir les règles métier exprimées en « Orienté Objet ».  
Le répertoire « Infrastructure » permettra de faire la relation entre la base de données et les objets « Domain ». En règle générale, il va contenir ce qui permet de faire les relations entre le monde extérieur et le « Domain » métier.

Vous allez créer une « class » « BasketService » dans un répertoire « Infrastructure » dont le rôle sera de retourner les objets métiers « Domain » à partir des structures d’informations d’entrée. Elle va récupérer les informations depuis la base de données et réaliser le mapping des données avec les objets métier du « Domain ». Il s’agit d’une « Indirection » (pattern GRASP), c’est une classe intermédiaire. Dans un premier temps, vous pouvez utiliser dans « BasketService.cs » la procédure existante qui charge les données depuis le ficher JSON. N’oublier pas d’enlever la notion de « static » qui n’a pas de sens en Orienté Objet.

|  |
| --- |
| namespace Basket.Infrastructure  {  public class BasketService  {  public Domain.Basket GetBasket(IList<BasketLineArticle> basketLineArticles)  {  // here your code implementation  }  }  } |

**Basket/Infrastructure/BasketService.cs**

Vous vous en doutez probablement, un panier (« Basket » en anglais) est constitué de ligne d’articles qui sont liés à un article et un nombre d’articles. A vous d’appliquer le cours et de créer le « Domain » model associé. Chercher et modéliser qui est responsable de qui et de quelles règles.

Votre « Domain » et votre classe « BasketService » devront pouvoir être utilisés par la classe « BasketOperation. » ci-dessous que vous devez ajouter à votre projet. Cette classe « BasketOperation » fera ainsi office de « Pattern Controleur » à l’utilisation des opérations du panier, pas besoin de savoir ce qu’il y a en dessous pour l’utiliser, il masquera la complexité.

|  |
| --- |
| namespace Basket  {  public class BasketOperation  {  private readonly Infrastructure.BasketService \_basketService;  public BasketOperation(Infrastructure.BasketService basketService)  {  \_basketService = basketService;  }  public int CalculateAmout(IList<BasketLineArticle> basketLineArticles)  {  Domain.Basket basket = \_basketService.GetBasket(basketLineArticles);  return basket.CalculateAmount();  }  }  } |

**Basket/BasketOperation.cs**

Le test code du test doit lui aussi être mis à jour comme ci-dessous afin d’utiliser notre « Controller ».

|  |
| --- |
| [TestMethod]  [DynamicData("Baskets")]  public void ReturnCorrectAmoutGivenBasket(BasketTest basketTest)  {  var basKetService = new BasketService();  var basketOperation = new BasketOperation(basKetService);  var amountTotal = basketOperation.CalculateAmout(basketTest.BasketLineArticles);  Assert.AreEqual(amountTotal, basketTest.ExpectedPrice);  } |

**BasketTest/BasketOperation\_CalculateBasketAmoutShould.cs**

A vous de réaliser tout le reste du code afin que le test passe au vert.

## Intégration continue

La mise en place d’une build « via un serveur de build » va vous aider à savoir que le code que vous poussez sur git est un code qui fonctionne (compilation, etc.). C’est-à-dire que vous allez être sûr que l’application que vous développez fonctionne ailleurs que sur votre ordinateur. Par exemple : cela vous permet de savoir que les sources positionnées sur git sont complètes (qu’il ne manque pas de fichiers).

Pour ce TP, nous allons utiliser « travis » comme serveur d’intégration continue qui est gratuit pour les projets github publics.

Vous pouvez vous connecter sur le site :

<https://travis-ci.org>

Créez un compte en utilisant votre profile github et configurez une build.

Vous allez avoir besoin de positionner à la racine de votre git un fichier de configuration pour que « travis » sache quelles instructions jouer. Voici un exemple ci-dessous, à vous de l’adapter à votre code :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | language: csharp | |  | dist: trusty | |  | mono: none | |  | dotnet: 2.0.0 | |  | install: | |  | - dotnet restore ./src/Basket.sln | |  | script: | |  | - dotnet build ./src/Basket.sln | |  | - dotnet test ./src/BasketTest/BasketTest.csproj | |

**.travis.yml**

Configurer « travis » pour que chaque modification réalisée dans le git distant déclenche automatiquement une « build ».

## Code Objet => Injection de Dépendance

Les tests en l’état actuel ne sont pas unitaires car pour le moment le code fait appel à une ressource externe qui est le disque dur. De plus, le code présent dans « BasketService.cs » fait trop de chose en même temp (cohérence).

Vous allez remédier à cela. Sortez le code qui réalise l’appel à la base de données dans un fichier « ArticleDatabaseJson.cs » (c’est le pattern, pure invention)

|  |
| --- |
| namespace Basket.Infrastructure  {  public class ArticleDatabaseJson  {  public ArticleDatabase GetArticle(string id)  {  var codeBase = Assembly.GetExecutingAssembly().CodeBase;  var uri = new UriBuilder(codeBase);  var path = Uri.UnescapeDataString(uri.Path);  var assemblyDirectory = Path.GetDirectoryName(path);  var jsonPath = Path.Combine(assemblyDirectory, "article-database.json");  IList<ArticleDatabase> articleDatabases =  JsonConvert.DeserializeObject<List<ArticleDatabase>>(File.ReadAllText(jsonPath));  var article = articleDatabases.First(articleDatabase => articleDatabase.Id == id);  return article;  }  }  } |

**Basket/Infrastructure/ArticleDatabaseJson.cs**

Mettez à jour le reste du code afin que ce nouvel objet soit Injecté dans la classe « BasketService », comme ci-dessous.

|  |
| --- |
| [TestMethod]  [DynamicData("Baskets")]  public void ReturnCorrectAmoutGivenBasket(BasketTest basketTest)  {  var basKetService = new BasketService(**new ArticleDatabaseJson()**);  var basketOperation = new BasketOperation(basKetService);  var amountTotal = basketOperation.CalculateAmout(basketTest.BasketLineArticles);  Assert.AreEqual(amountTotal, basketTest.ExpectedPrice);  } |

**BasketTest/BasketOperation\_CalculateBasketAmoutShould.cs**

Vérifier que vos tests sont toujours passants. Ce qui est normalement le cas. Maintenant que l’objet qui va rechercher les données en base de données est injecté, on va pourvoir le remplacer par une implémentation « mockée » afin d’avoir un vrai test unitaire complètement indépendant de toute ressource Physique.

Pour cela extrayez une interface depuis la classe « ArticleDabaseJson ».

|  |
| --- |
| namespace Basket.Infrastructure  {  public interface IArticleDatabase  {  ArticleDatabase GetArticle(string id);  }  } |

**Basket/Infrastructure/IArticleDatabase.cs**

Utilisez partout cette interface plutôt que directement le type « ArticleDabaseJson ». Les tests doivent toujours rester verts.

Maintenant vous pouvez créer un nouvel objet « ArticleArticleDatabaseMock » qui implémente cette interface et utilise le « mock » utilisé précédemment.

|  |
| --- |
| namespace Basket.Infrastructure  {  public class ArticleDatabaseMock : IArticleDatabase  {  public ArticleDatabase GetArticle(string id)  {  switch (id)  {  case "1":  return new ArticleDatabase { Id = "1", Price = 1, Stock = 35, Label = "banana", Category = "food" };  case "2":  return new ArticleDatabase  {  Id = "1",  Price = 500,  Stock = 20,  Label = "frigo",  Category = "electronic"  };  case "3":  return new ArticleDatabase { Id = "1", Price = 1, Stock = 68, Label = "chaise", Category = "desktop" };  default:  throw new **NotImplementedException**();  }  }  }  } |

**Basket/ArticleDatabaseMock.cs**

Votre test doit maintenant fonctionner quelque soit l’implémentation injectée. L’implémentation « mock » permet à votre test de ne pas être lié à aucune ressource Physique. Il est dans ce cas complètement unitaire. En faisant de l'injection de dépendance, vous avez suivi le pattern GRAPS de la « protection des variations »; votre code est protégé des variations.

|  |
| --- |
| [TestMethod]  [DynamicData("Baskets")]  public void ReturnCorrectAmoutGivenBasket(BasketTest basketTest)  {  var basKetService = new BasketService(**new ArtcileDatabaseMock()**);  // var basKetService = new BasketService(**new ArticleDatabaseJson()**);  var basketOperation = new BasketOperation(basKetService);  var amountTotal = basketOperation.CalculateAmout(basketTest.BasketLineArticles);  Assert.AreEqual(amountTotal, basketTest.ExpectedPrice);  } |

**BasketTest/BasketOperation\_CalculateBasketAmoutShould.cs**

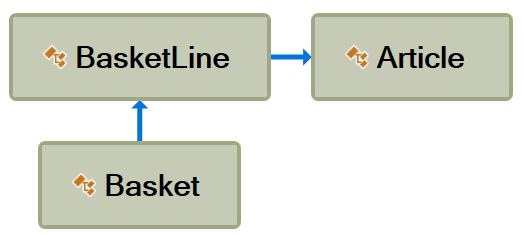
## Code Objet => Mise en place du polymorphisme

Si vous avez bien réalisé les étapes précédentes, vous devez avoir une classe « Article » dans votre « Domain » qui ressemble plus ou moins à la représentation ci-dessous :

|  |
| --- |
| namespace Basket.Domain  {  public class Article  {  public Article(string id, int price, string category)  {  Id = id;  Category = category;  Price = price;  }  public int Price { get; }  public string Id { get; }  public string Category { get; }  public int CalculateAmout()  {  switch (Category)  {  case "food":  return Price \* 100 + Price \* 12;  case "electronic":  return Price \* 100 + Price \* 20 + 4;  case "desktop":  return Price \* 100 + Price \* 20;  default:  throw new **NotImplementedException**();  }  }  }  } |

**Basket/Domain/Article.cs**

Et un digramme de dépendance qui ressemble à l’image ci-dessous :



**Diagramme de dépendance du « Domain »**

Vous allez ajouter un nouvel article dans la base de données. Dans une nouvelle catégorie « toy », avec les informations ci-dessous :

|  |
| --- |
| …  {  "Id": "4",  "Label": "Grumly",  "Category": "toy",  "Price": 39,  "Stock": 55  }  … |

**Basket/article-database.json**

Cet article possède une TVA à 20% et possède actuellement une remise immédiate de 3€. Attention la TVA ne s’applique que sur le prix remisé. Par exemple :

* Si vous achetez 2 Grumlys, le montant total retourné sera de « 8640 »

**Mettez à jour les tests en conséquence puis réalisez l’implémentation.**

Il y a un problème avec cette classe « Article », car elle possède des règles qui sont jouées en fonction de la propriété « Catégorie ». Elle est de moins en moins cohérente. Dans notre exemple, il n’y a actuellement que quatre règles très simples. Dans la réalité il y en aurait plus d’une centaine avec des règles très spécifiques en fonction de beaucoup plus de paramètres. La cohérence de l’objet « Article » en l’état actuel serait très faible.

Dans ce cas, pour résoudre cette problématique, vous allez utiliser le « Polymorphisme ». Vous l’avez plus ou moins déjà utilisé en réalisant précédemment de l’injection de dépendance. Créez une classe abstraite « ArticleBase » et les quatre implémentations associées. Soyez paresseux, créez cette classe à l’aide des raccourcis. C’est l’éditeur de texte qui doit tout générer pour vous.

|  |
| --- |
| Namespace Basket.Domain  {  public abstract class ArticleBase  {  public ArticleBase(string id, int price)  {  Id = id;  Price = price;  }  public int Price { get; }  public string Id { get; }  public abstract int CalculateAmout();  }  } |

**Basket/Domin/ArticleBase.cs**

Exemple d’implementation “ArticleFood”:

|  |
| --- |
| namespace Basket.Domain  {  public class ArticleFood : ArticleBase  {  public ArticleFood(string id, int price) : base(id, price)  {  }  public override int CalculateAmout()  {  return Price \* 100 + Price \* 12;  }  }  } |

**Basket/Domin/ArticleFood.cs**

**Mettez à jour tout le code afin que les tests soient toujours passants.** La classe article devrait ainsi ne plus exister et le « switch » présent dans celle-ci, déplacée.

## Intégration continue => Couverture de code

Ajouter les lignes ci-dessous à votre projet de test :

|  |
| --- |
| <Project Sdk="Microsoft.NET.Sdk">  <PropertyGroup>  <TargetFramework>netcoreapp2.0</TargetFramework>  <IsPackable>false</IsPackable>  <DebugType>full</DebugType>  </PropertyGroup>  <ItemGroup>  <PackageReference Include="coverlet.msbuild" Version="1.2.0" />  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.7.0" />  <PackageReference Include="MSTest.TestAdapter" Version="1.3.0" />  <PackageReference Include="MSTest.TestFramework" Version="1.3.0" />  </ItemGroup>  <ItemGroup>  <ProjectReference Include="..\Basket\Basket.csproj" />  </ItemGroup>  <ItemGroup>  <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="2.0.0" />  </ItemGroup>  </Project> |

**BasketTest/BasketTest.csproj**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | language: csharp | |  | dist: trusty | |  | mono: none | |  | dotnet: 2.0.0 | |  | install: | |  | - dotnet restore ./src/Basket.sln | |  | script: | |  | - dotnet build ./src/Basket.sln | |  | - dotnet test ./src/BasketTest/BasketTest.csproj /p:CollectCoverage=true /p:CoverletOutputFormat=opencover | |

**.travis.yml**

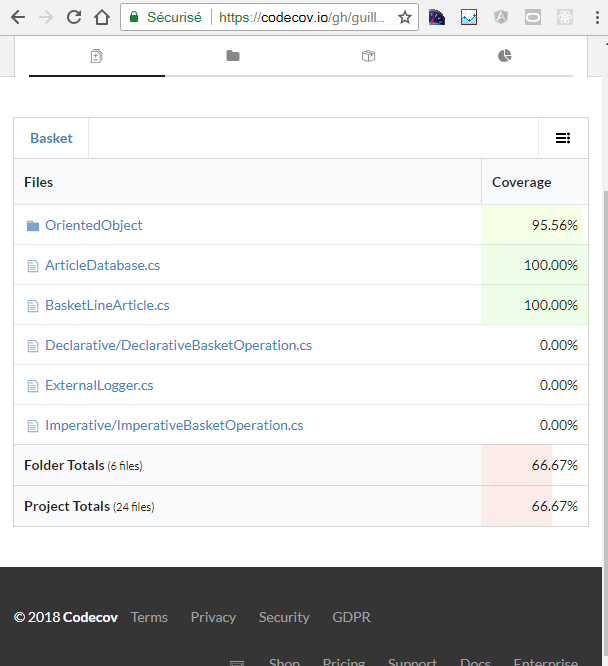
(Optionnelle) Si vous le souhaitez (vous pouvez configurer « travis » afin que le serveur publie la couverture de code sur le site : <https://codecov.io>

Il faudra rajouter à la fin de votre fichier les lignes suivantes :

|  |  |
| --- | --- |
|  | after\_script: |
|  | - bash <(curl -s https://codecov.io/bash) |

**.travis.yml**

Cela vous permettra de « visualiser » de manière détaillée le taux de couverture de votre code.



## Ajout d’une méthode à notre contrôleur « opération »

Nous souhaitons ajouter à notre classe « BasketOperation » qui fait office de « pattern contrôleur » une méthode qui retourne le prix courant d’une seule ligne d’un article. Pour cela ajoutez un nouveau test unitaire.

|  |
| --- |
| [TestMethod]  public void ReturnCorrectAmoutGivenBasketLine()  {  var basketLineArticle = new BasketLineArticle {Id = "4", Number = 2, Label = "Grumy"};  var basKetService = new BasketService(new ArticleArticleDatabaseMock());  var basketOperation = new BasketOperation(basKetService);  var amountTotal = basketOperation.CalculateAmout(basketLineArticle);  Assert.AreEqual(amountTotal, 8640);  } |

**BasketTest/BasketService\_CalculateBasketAmoutShould.cs**

**A vous de réaliser l’implémentation du code pour que ce test fonctionne.**

## Protection des variations => Mise en place d’un logger

Vous souhaitez logger en mode « Information » les appels de chaque méthode de « BasketOperation ». Les messages :

* Appel total panier réalisé
* Appel total ligne panier réalisé

Vous souhaitez aussi logger depuis le fichier « BasketService.cs » spécifiquement un message « problème lors de la récupération des données article », si une exception est lancée. L’exception ne doit pas être arrêtée, elle doit être relancée car si la base n’est pas disponible, le panier ne peut tout simplement pas s’afficher correctement (c’est un plantage).

Pas de chance ; on vous impose le fait d’utiliser un logger, issue d’une librairie externe qui a l’air vraiment peu performante et peu testable, car couplé à une ressource externe (disque dur).

Ajoutez ce fichier qui fera office de librairie externe de log mal fichu.

|  |
| --- |
| namespace Basket  {  public static class ExternalLogger  {  private static void Write(string message)  {  var codeBase = Assembly.GetExecutingAssembly().CodeBase;  var uri = new UriBuilder(codeBase);  var path = Uri.UnescapeDataString(uri.Path);  var assemblyDirectory = Path.GetDirectoryName(path);  string filePath = Path.Combine(assemblyDirectory, "log.txt");  File.AppendAllText(filePath, message, Encoding.UTF8);  }  public static void LogInformation(string message)  {  Thread.Sleep(300);  Write($"Information: {message}{Environment.NewLine}");  }  public static void LogError(Exception exception,string message)  {  Thread.Sleep(700);  Write($"Error: {message} {exception}");  }  }  } |

**Basket/ExternalLogger.cs**

Vous souhaitez que votre code soit protégé de cette librairie (minimum de dépendance/couplage, possibilité de changer très facilement de logger sans que cela impacte votre code).

**A vous de réaliser le code et le test associé le plus proprement que possible.**

## Nouvelle règle => TDD (Test Driven Developement)

Pour 20 bananes achetées, la 21ème est gratuite (cette règle est valable pour tous les aliments). Par exemples :

* Pour 21 bananes dans le panier, le prix total du panier est de 22,40€
* Pour 43 bananes dans le panier, le prix total du panier est de 45,92€

**A vous d’ajouter le nouveau test unitaire associé puis le code respectant les rôles et responsabilités des objets. N'hésitez pas à refaire du polymorphisme si nécessaire avec les objets "BasketLine" et à adapter/refactorer le code.**

## Nouvelle règle => TDD (Test Driven Development)

Si le stock de jouets est inférieur strictement à 50 alors le prix du jouet augmente de 10 euros.  
Par exemple :

* S’il y a 55 articles en stock et que vous en avez 10 dans le panier, la règle s’applique sur 5 de vos articles du panier uniquement, le prix du panier est alors de 472,00€.

**A vous d’ajouter le nouveau test unitaire associé puis le code respectant les rôles et responsabilités des objets.**

## Nouvelle règle => TDD (Test Driven Development)

Si vous achetez 8 articles de la catégorie « desktop » alors un jouet est gratuit. Cette offre ne fonctionne qu'une seule fois (sur 1 jouet).

Par exemple :

* Pour 8 chaises dans le panier, ainsi que 2 Grumlys, le prix total du panier est de 403,68€.

**A vous d’ajouter le nouveau test unitaire associé puis le code respectant les rôles et responsabilités des objets.**

## Code Objet => Mise en place d’une « pure invention »

Le switch a normalement été déplacé dans la « class » « BasketService » et permet maintenant d’instancier le bon type d’article à chaque fois. C’est un bon use case pour mettre en place le pattern « factory » dont le seul rôle sera d’instancier le bon « Article » en fonction de la « Categorie ».

**Créer une méthode qui se nomme « ArticleFactory » et qui aura une méthode publique « Create » qui retournera un « ArticleBase ».**

Le vrai intérêt d’une factory n’est pas visible dans ce TP, elle permet d’éviter la duplication de « if » et de « switch » identique, dans le code. On utilise la factory qui centralise ces éléments logiques. Ce pattern est un pattern indispensable aux applications. Elle va tout de même fortement vous aidez pour la question suivante !

## Nouvelle règle => TDD (Test Driven Development)

Pendant le mois de décembre de chaque année :

* Le prix de la nourriture double.
* Le prix des appareils de bureau augmente de 10€.
* Le prix des jouets augmente de 3€.

Pour information "System.DateTime" est le type qui permet de gérer les dates en dotnet.

|  |
| --- |
| var dateTime = new Datetime(2018, 12, 1); // correspond au 1 Décembre 2018  Console.WriteLine(dateTime.Month); // affiche: 12 |

**Exemple manipulation de date**

Par exemple :

* Pour 12 bananes, 2 Grumlys et 4 chaises, le prix total pendant le mois de Décembre est de 403,68€

Pendant le mois de Janvier de chaque année :

* Le prix des appareils électroniques dont le prix est supérieur strictement à 300€ diminue de 30€.
* Le prix des appareils de bureau dont le prix est supérieur strictement à 20€ diminue de 5€.

Par exemple :

* Pour 1 frigo et 4 chaises, le prix total pendant le mois de Janvier est de 775,24€

**A vous d’ajouter les nouveaux tests unitaires associés puis le code respectant les rôles et responsabilités des objets.**