

R3.04 - QUALITE DE DEVELOPPEMENT

CM2 : Software patterns, Tests

Vincent COUTURIER

Bonnes pratiques

1. Choisir la bonne méthode de développement (**CM1**)
2. Modéliser (analyse, conception) (**CM1**)
3. Choisir la bonne architecture de l'application (et appliquer des patrons d'architecture) (**CM1**)
4. Appliquer des patrons logiciels (software patterns) (**CM2**)
5. Développement (**CM1**) :
 - a. Créer des vues accédant aux tables de la base de données
 - b. Eventuellement des procédures/fonctions stockées
 - c. ORM (Object-Relational Mapping)
 - d. Utiliser un framework
 - e. Gestion des exceptions
 - f. Documenter
6. Versioning (**CM1**)
7. Intégration continue / Déploiement / Usine logicielle / DevOps (**CM1**)
8. Tests (**CM2**)



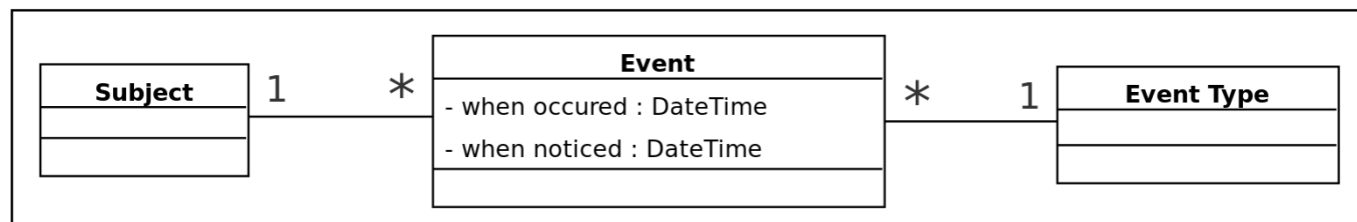
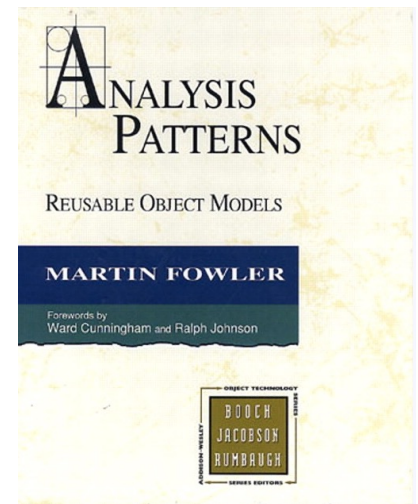
APPLIQUER DES PATRONS LOGICIELS (SOFTWARE PATTERNS)

Patron logiciel ?

- Un patron est un « guide de bonne pratique présentant une solution éprouvée dédiée à résoudre un problème fréquemment rencontré » lors de :
 - L'analyse => *Patrons de domaine* (« Analysis patterns »)
 - La définition de l'architecture (début de la conception) => *Patrons d'architecture* (« Architectural patterns »). Cf. slides précédents.
 - La conception (détaillée) => *Patrons de conception* (« Design patterns »)
 - La programmation => *Patrons d'implémentation* ou *Idiomes* (« Idioms »).

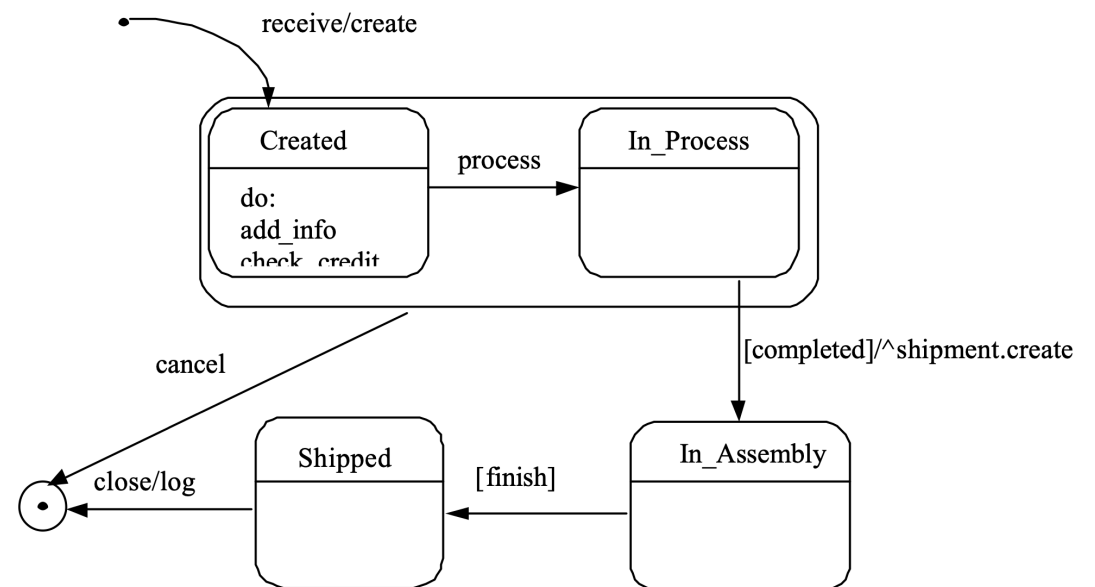
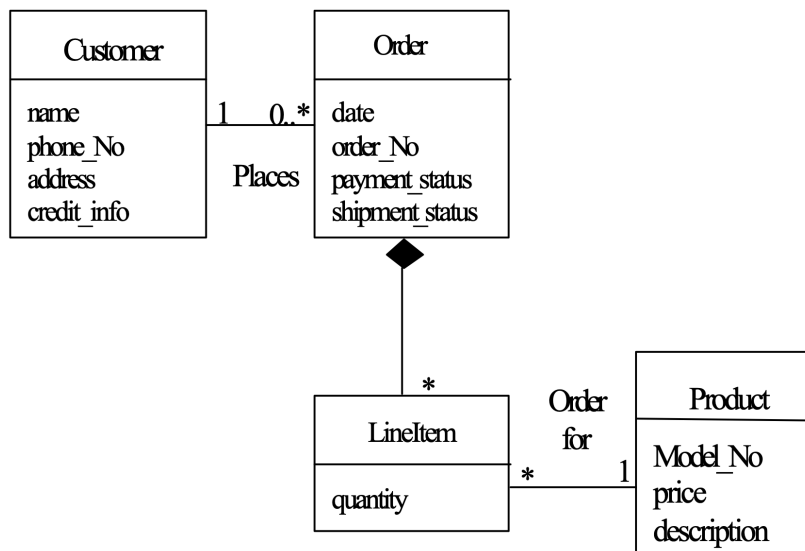
Patrons d'analyse

- Souvent, modélisations objet (diagramme de classes d'analyse) dédiées à modéliser un problème d'un domaine spécifique (financier, vente, etc.) ou un problème générique
- La bible :
 - <http://uml2.narod.ru/files/docs/13/AnalysisPatterns.pdf>
- Exemple :
 - Patron d'analyse générique modélisant l'abonnement à un évènement



Patrons d'analyse

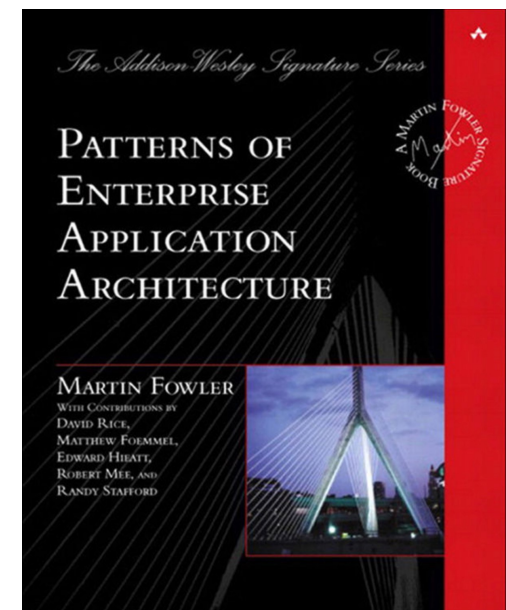
- Exemple :
 - Patron d'analyse non générique dédié à modéliser les commandes (« Order pattern »)



https://www.researchgate.net/publication/250165865_Analysis_Patterns_for_the_Order_and_Shipment_of_a_Product

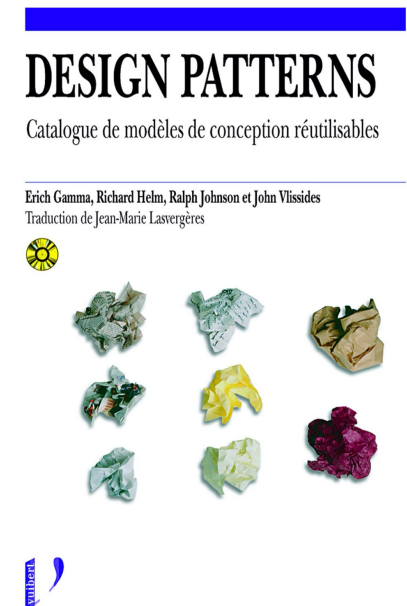
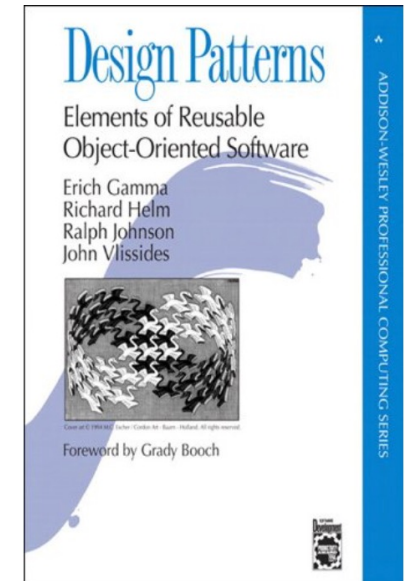
Patron d'architecture

- = modèle de référence qui sert de source d'inspiration lors de la conception de l'architecture d'un système ou d'un logiciel informatique en sous-éléments plus simples.
- Exemples :
 - MVC
 - MVC2
 - Modèle Vue Présentateur (MVP ou MVA)
 - Présentation-Modèle (PM)
 - MVVM
 - ...
 - Plus de détails :
 - <https://frebourg.es/architectures-applicatives/>
 - <https://www.linkedin.com/pulse/understanding-difference-between-mvc-mvp-mvvm-design-rishabh-software/>



Patron de conception

- = « *Arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception (détaillée) d'un logiciel* »
- Décrit une solution standard, utilisable dans la conception de différents logiciels et lors de leur implémentation => Patrons de conception procédurale, Patrons de conception objet, etc.
- Bible : « Design Patterns: Elements of Reusable Software », co-écrit par 4 auteurs : Gamma, Helm, Johnson et Vlissides (Gang of Four - GoF; en français la bande des quatre) en 1994.
 - Ce livre, devenu un best-seller, décrit 23 patrons à appliquer lors de la conception et qui se retrouveront dans le code ainsi que la manière de s'en servir.



Patron de conception

Singleton	
-	<u>singleton</u> : Singleton
-	Singleton()
+	<u>getInstance()</u> : Singleton

- Ex. de design pattern du GoF : Singleton
 - On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà.

```
public class Singleton
{
    private static Singleton _instance;
    static readonly object instanceLock = new object();

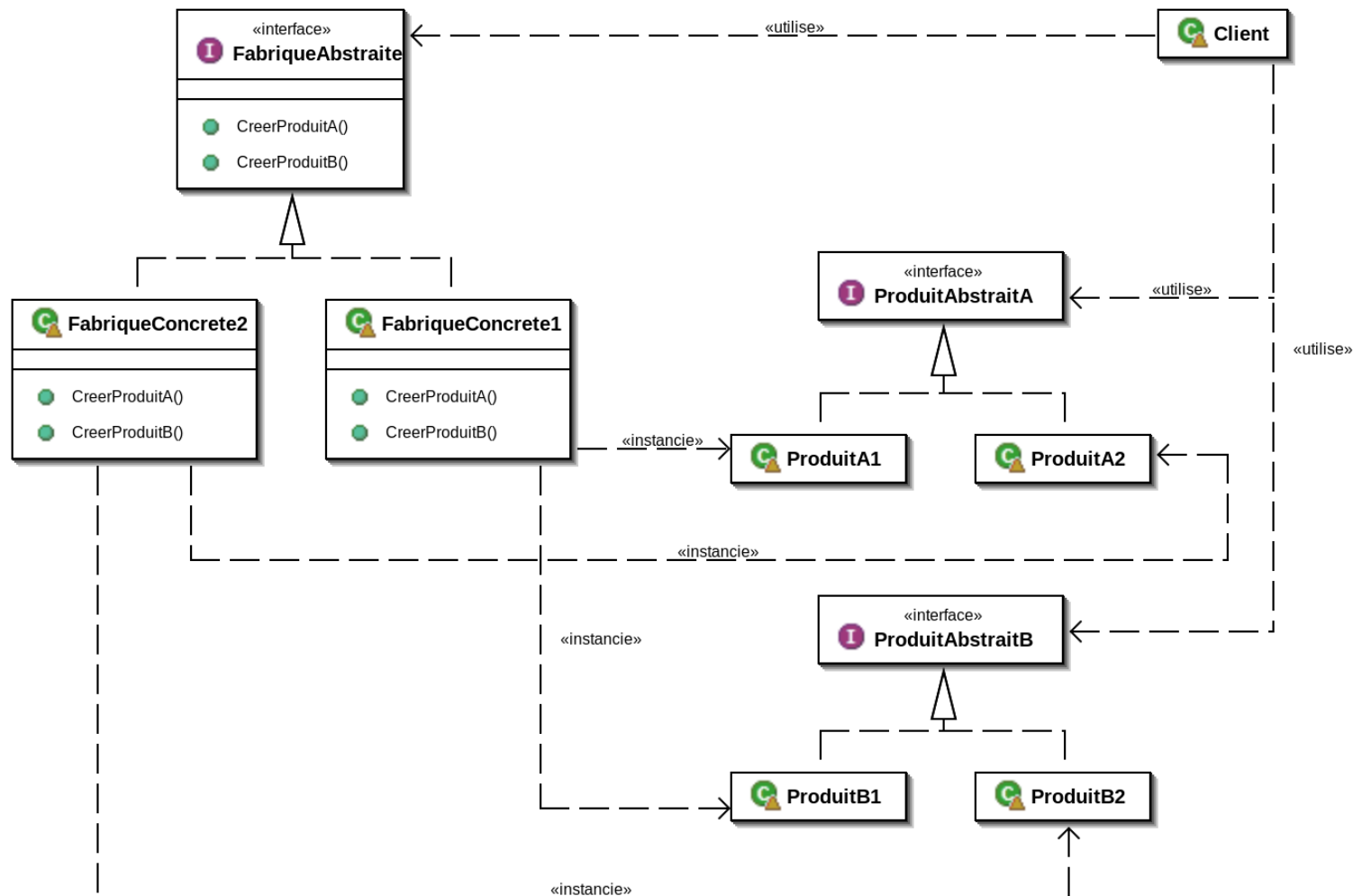
    private Singleton()
    {
    }

    public static Singleton getInstance()
    {
        if (_instance == null) //Les locks prennent du temps, il est préférable de vérifier d'abord la nullité de l'instance.
        {
            lock (instanceLock)
            {
                if (_instance == null) //on vérifie encore, au cas où l'instance aurait été créée entretemps.
                {
                    _instance = new Singleton();
                }
            }
        }

        return _instance;
    }
}
```

Patron de conception

- Ex. de design pattern du GoF : Abstract Factory
 - Fournit une interface pour créer des familles d'objets liés ou interdépendants sans avoir à préciser au moment de leur création la classe concrète à utiliser



Patron de conception

- Ex. de design pattern du GoF : Abstract Factory en C#

```
abstract class GUIFactory {
    public static GUIFactory getFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys==0) {
            return(new WinFactory());
        } else {
            return(new OSXFactory());
        }
    }
    public abstract Button createButton();
}

class WinFactory:GUIFactory {
    public override Button createButton() {
        return(new WinButton());
    }
}

class OSXFactory:GUIFactory {
    public override Button createButton() {
        return(new OSXButton());
    }
}
```

Patron de conception

```
abstract class Button {
    public string caption;
    public abstract void paint();
}

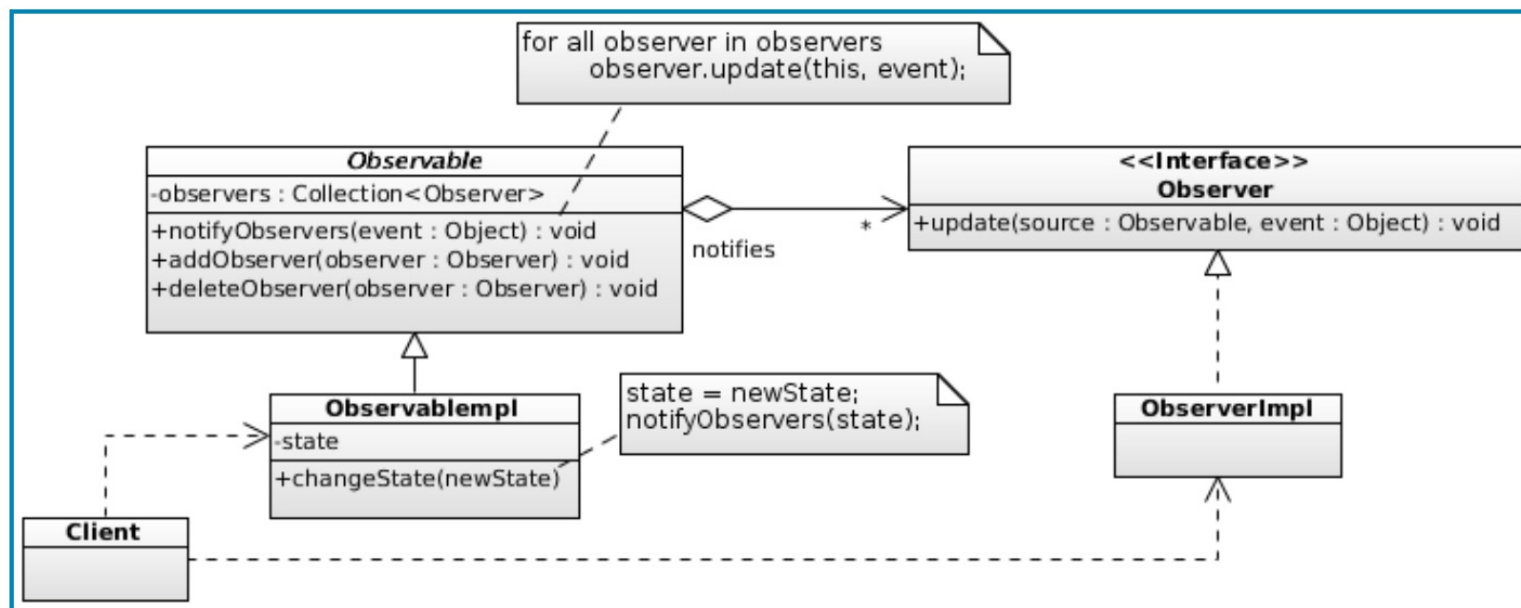
class WinButton:Button {
    public override void paint() {
        Console.WriteLine("I'm a WinButton: "+caption);
    }
}

class OSXButton:Button {
    public override void paint() {
        Console.WriteLine("I'm a OSXButton: "+caption);
    }
}

class Application {
    static void Main(string[] args) {
        GUIFactory aFactory = GUIFactory.getFactory();
        Button aButton = aFactory.createButton();
        aButton.caption = "Play";
        aButton.paint();
    }
    //output is
    //I'm a WinButton: Play
    //or
    //I'm a OSXButton: Play
}
```

Patron de conception

- Ex. de design pattern du GoF : Observer
 - Fournit un moyen pour tous les objets intéressés (*observers*) d'être notifiés par un objet observé (*observable*). En cas de notification, les *observateurs* effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les *observables*).
 - À utiliser lorsque :
 - Les changements d'état dans un ou plusieurs objets doivent déclencher le comportement d'autres objets.
 - Vous devez fournir des capacités de diffusion d'événements (broadcast).
 - Pattern mis en place notamment dans les *Observable Collections* (Binding C#)



Idiome

- = Construction spécifique à un langage de programmation, qui est une manière usuelle de mettre en œuvre une solution à un problème dans ce langage de programmation.
- L'utilisation d'un idiotisme par le programmeur lui évite d'avoir à remettre en question la structure détaillée du programme et améliore la qualité du produit.
- Ex. :
 - `for (i=0; i<100; i++)`
 - `i=i+1; -> i++;`

Idiomes C#

- Examples :

```
for (int i=0;i<list.Count; i++) {  
    DoSomething(list[i]);  
}
```

->

```
foreach (Item item in list) {  
    DoSomething(item);  
}
```

```
List<int> list2 = new List<int>();  
for (int num in list1) {  
    if (num>100) list2.Add(num);  
}
```

-> Expression Lambda

```
var list2 = list1.Where(num=>num>100);
```

Idiomes C#

- **Exemple :**

```
StreamReader sr = File.OpenText(path);  
string content = sr.ReadToEnd();  
sr.Close();
```

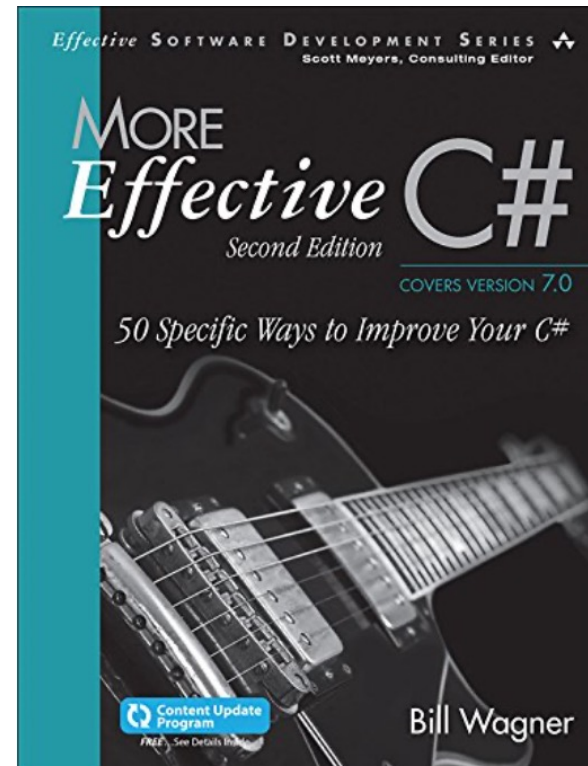
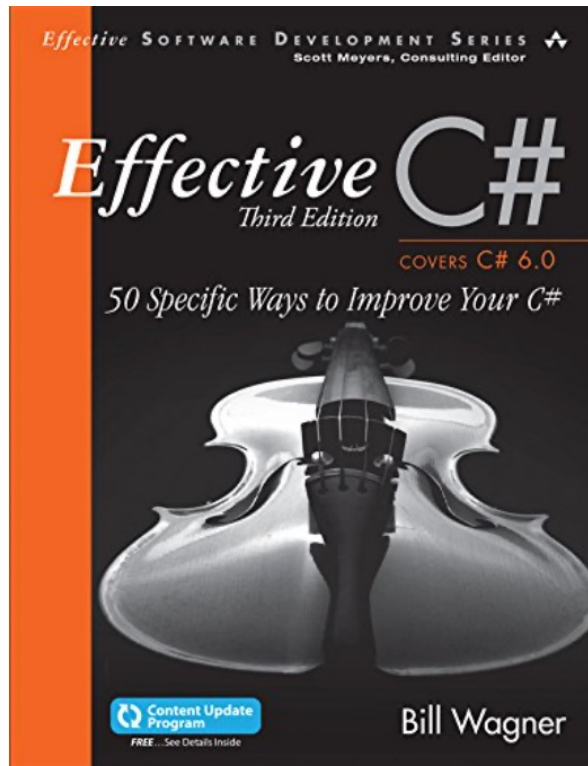
->

```
string content;  
using (StreamReader sr = File.OpenText(path)) {  
    content = sr.ReadToEnd();  
}
```

- **Exemple (C# 8.0) :**

```
List<int> numbers = null;  
(numbers ??= new List<int>()).Add(5);  
//??= affecte la valeur de son opérande droit à  
son opérande de gauche uniquement si l'opérande de  
gauche est évalué à null
```


La bible des Idioms C# (mais pas que)

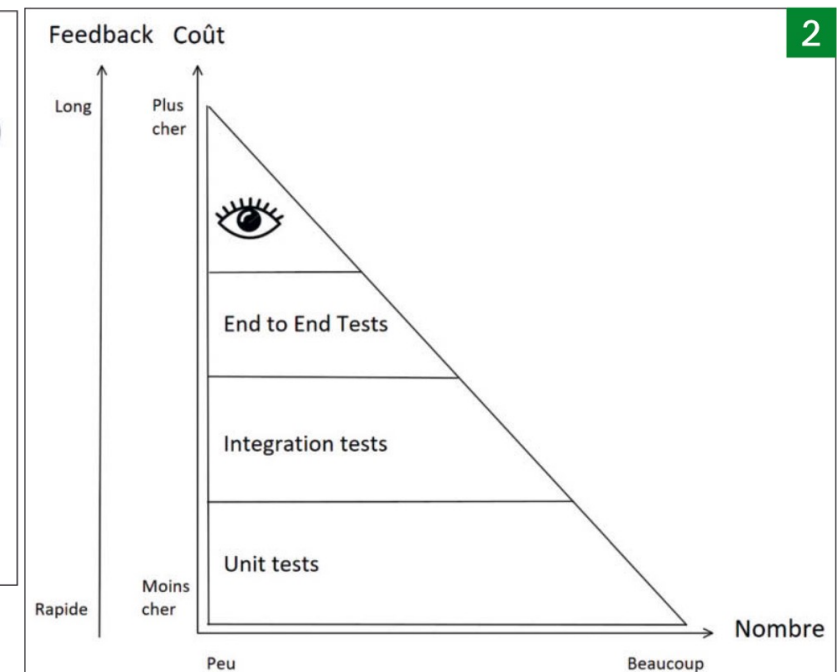
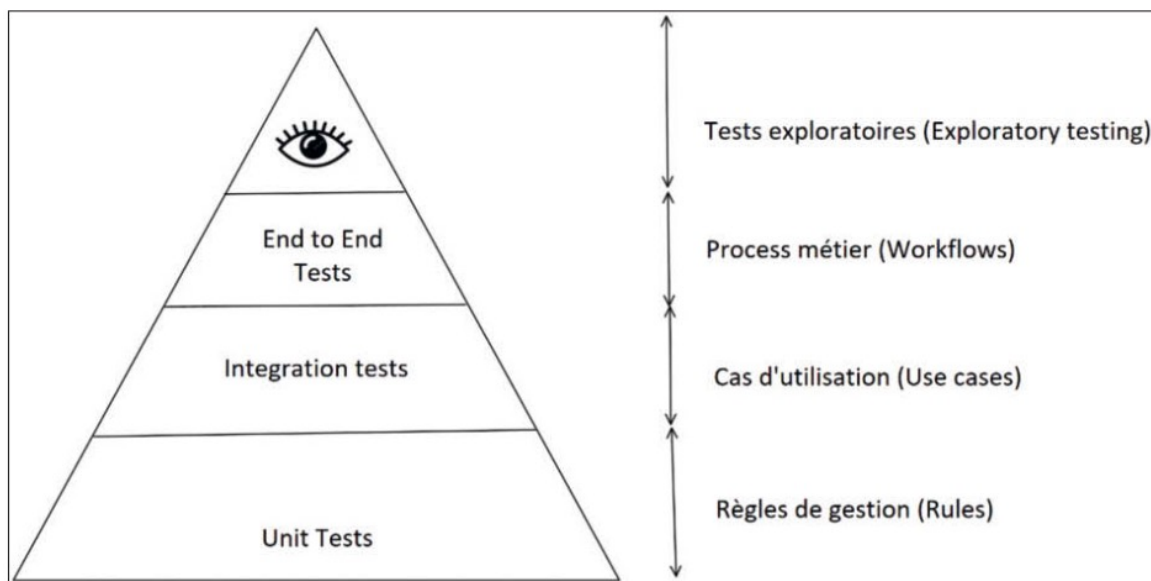




TESTER

Pyramide des tests

- C'est **PRINCIPALEMENT** au développeur de faire les tests et non à l'utilisateur !!!!!
- Pyramide des tests (Martin Fowler) :
 1. Tests unitaires : représentent généralement les règles de gestion de l'application
 2. Tests d'intégration : représentent les use-cases de l'application,
 3. Tests end-to-end : représentent les workflows ou process métier de l'application
 4. Tests utilisateurs : représentés par l'œil qui sont les tests manuels exploratoires de l'application



Tests unitaires (Unit Tests – UT)

- **Les + importants** : « permettent de valider le comportement d'une unité de code. La définition d'une unité de code est propre à chacun : une classe ou une fonctionnalité qui peut correspondre à plusieurs classes »
- = Test de composant, Test de module
- **A faire au fur et à mesure du codage de l'application**
- Dans les applications non critiques, l'écriture des tests unitaires a longtemps été considérée comme une tâche secondaire. Cependant, les méthodes Extreme programming (XP) ou Test Driven Development (TDD) ont remis les tests unitaires, appelés « **tests du programmeur** », au centre de l'activité de programmation.
- Les tests peuvent être manuels ou **MIEUX automatisés** :
 - Ecriture du code de test sans framework
 - Ou avec un framework comme xUnit (**BEAUCOUP MIEUX !**). Exemples :
 - cppunit, Google Test¹⁵ et Boost Test¹⁶ pour C++
 - CUnit pour C
 - JUnit, QUnit et Unit.js pour JavaScript
 - JUnit et TestNG pour Java
 - NUnit et MSTests/MSTestsv2 pour .NET
 - PHPUnit pour PHP

Tests unitaires : critères FIRST, règle des 3A

- **Un UT doit respecter les critères FIRST :**

- *Fast* : doit être rapide. On peut le lancer à tout moment sans que ça prenne du temps (qq secondes).
- *Isolated / Independent* : il ne faut pas qu'il y ait un ordre d'exécution des tests. Un test doit être indépendant de tous les tests qui l'entourent. Et une seule cause doit être à l'origine de son échec et non plusieurs.
- *Repeatable* : un test doit être déterministe. Si on le lance plusieurs fois avec une même donnée en entrée, le résultat ne doit jamais changer.
- *Self validating* : le test doit faire la validation lui-même (pas besoin d'intervention manuelle pour savoir s'il est passé ou a échoué).
- *Timely / thorough* : le test doit être minutieux, précis par rapport à ce qu'il teste. Le test doit être écrit au moment opportun. En appliquant la méthode TDD (*Test Driven Development*), on va écrire le test avant le code de production et donc l'écriture sera au moment où on en a besoin. Le test peut être écrit après l'écriture du code dans le cadre d'un code legacy (code sans test).

- **Un UT doit respecter la règle des 3A (Arrange/Act/Assert) :**
Cf. slide suivant

Tests unitaires en C#

- NUnit ou MSTests/MSTestsv2 :
 - NUnit suit le framework xUnit =>
 - Pratiquement les mêmes classes que pour PHP ou Java
 - Nécessite d'importer un package Nuget
 - MSTests/MSTestsv2 :
 - Framework de test plus récent, mais aussi plus complet. Ne respecte donc pas totalement NUnit
 - Les classes sont intégrées dans le framework .Net et donc directement utilisables (pas d'import de package).

Tests unitaires en C# avec MSTests/MSTestsv2

- Un test unitaire pour chaque méthode à tester
- Un test unitaire contient toujours 3 parties (AAA) : *Arrange* – *Act* – *Assert* ou en français *Arranger* – *Agir* – *Auditer*.
 - *Arrange* : préparation des variables : variables passées en paramètres, variable contenant la valeur espérée, variable contenant la valeur retournée, etc.
 - *Act* : appel de la méthode à tester
 - *Auditer* : vérification du test. Utilisation de la (ou des) classe `Assert` et de ses méthodes. Exemples :
 - `Assert.AreEqual(valesperree, valretournee)` : pour tester la valeur retournée et celle espérée
 - `Assert.AreNotEqual` : inverse
 - `Assert.AreSame(objet1, objet2)` : Vérifie que deux variables objets spécifiées font référence au même objet.
 - `Assert.AreNotSame` : inverse
 - `Assert.IsNull(objet)` : Vérifie que l'objet spécifié est **null**.
 - `Assert.IsNotNull` : inverse
- On réalise au moins 1 test qui réussit et 1 qui échoue et donc renvoie une exception.

Tests unitaires en C# avec MSTests/MSTestsv2

- Exemple :

- Code à tester :

```
public class Account {  
    double solde;  
    public double Solde{ get { return solde; } set { solde=  
value; } }  
    public void Retrait(double montant) {  
        if(Solde >= montant) {  
            Solde -= montant;  
        }  
        else  
        {  
            throw new ArgumentException("Retrait excède le  
solde!");  
        }  
    }  
    ...  
}
```


Tests unitaires en C# avec MSTests/MSTestsv2

- Exemple :
 - Test qui réussit :

```
[TestMethod]
public void Retrait_MontantValide_ModifSolde() {
    // arrange
    double soldecourant= 10.0;
    double montantretrait= 1.0;
    double montantespere= 9.0;
    Account compte= new Account(soldecourant);
    // act
    compte.Retrait(montantretrait);
    double actuel = compte.Solde;
    // assert
    Assert.AreEqual(montantespere, actuel, "Test non OK");
}
```

Tests unitaires en C# avec MSTests/MSTestsv2

- Exemple :
 - Test qui lève une exception :

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void Retrait_MontantSupSolde_Exception() {
    // arrange
    Account compte = new Account(10.0);
    // act
    compte.Retrait(20.0);
    // assert is handled by the ExpectedException
}
```

Tests unitaires en C# avec MSTests/MSTestsv2

- Film 1^{er} test : [film1](#)
- Film 2ème test : [film2](#)

Tests d'intégration (Integration Test – IT)

- D'après Martin Fowler, « un test d'intégration détermine si les composants d'une application continuent à fonctionner correctement quand on les connecte ensemble ».
- 2 types de tests d'intégration :
 - *Narrow Integration tests* (étroite couverture d'unités) : leur scope s'arrête à quelques composants tout en substituant d'autres dépendances (notion de substitution ou mock). Ces tests ressemblent à des tests unitaires au niveau de l'utilisation des frameworks et la façon de les écrire.
 - *Broad Integration tests* (couverture étendue d'unités) : tests qui vont avoir besoin d'instances réelles de services, potentiellement d'une base de données... L'idée est d'exécuter un scénario d'un plus large scope que celui utilisant les substituts (mocks). Ainsi on s'assure que l'intégration des composants se prépare au mieux pour éviter les surprises aux tests utilisateurs.

Tests d'intégration : format Gherkin

- Formalisme "given/when/then" qui structure et rapproche le test au langage humain.
- Facilite l'écriture des tests avec le PO ou les utilisateurs puisqu'il se rapproche d'une spécification ou d'une documentation.
- On l'utilise souvent dans le cadre d'une méthodologie *BDD (Behavior Driven Development)*.
- Exemples de scénarios :
 - Chaque ligne correspond à une "step definition" qui sera associée à une méthode (et donc une implémentation). Cette dernière sera exécutée quand l'étape (appelée aussi step) sera atteinte et ainsi elle détermine si le test a réussi ou a échoué.
 - Scenario : Convert amount if the rate is found
 - Given* I have 100 EUR
 - And I have entered USD as target currency
 - And the rate is 1.14
 - When* I convert amount
 - Then* the converted amount should be 114 USD
 - Scenario : Convert amount if the rate is not found
 - Given* I have 100 EUR
 - And I have entered TOG as unfound target currency
 - When* I convert amount
 - Then* I should get an error

Tests d'intégration : comment les mettre en place ?

- S'écrivent avec les mêmes outils que ceux des tests unitaires.
=> On peut utiliser les substituts pour tester un ou plusieurs composants et utiliser les implémentations de production pour d'autres.
- Si on ne veut pas utiliser les substituts alors que notre code doit accéder à une dépendance externe, on a plusieurs solutions :
 - Cas où on doit appeler une API par exemple :
 - Dans ce cas, il faut un environnement prêt avec la version à tester du service.
 - Cas où on a besoin d'accéder à une base de données.
 - Pour pouvoir réaliser ce test, il sera nécessaire d'insérer les données du test, le faire tourner et puis nettoyer les données afin qu'au lancement suivant le même test ne soit pas altéré.
 - Autre solution pour lancer le test sans devoir déployer une base : utiliser une bases in-memory (base en mémoire qui sera supprimée quand tous les tests auront fini leur exécution)
- Pour le format Gherkin: SpecFlow (<https://spec-flow.org>) pour .Net ou Cucumber (<https://cucumber.io/>) pour java.

Tests end-to-end (E2E Tests)

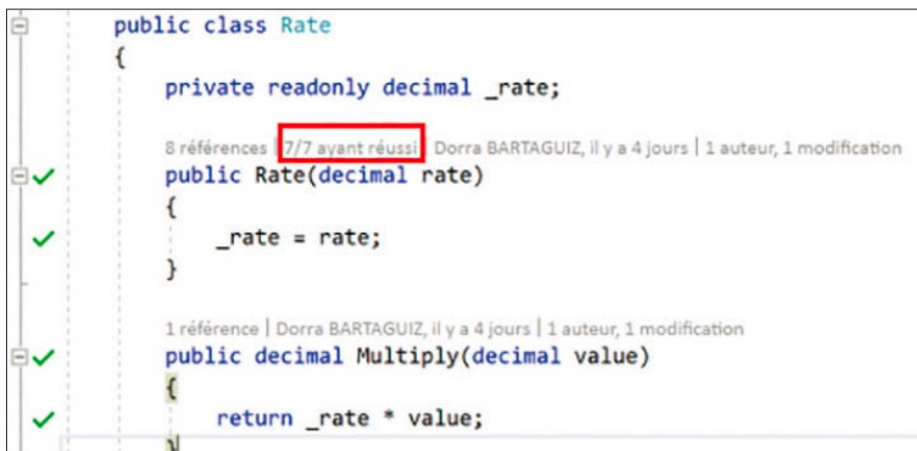
- Permettent de vérifier que l'ensemble des composants d'une fonctionnalité fonctionnent de bout en bout, de la partie IHM aux dépendances externes (base de données ou service externe comme une API).
- Auparavant, on les appelait « Tests d'IHM ».
- Sont utilisés pour les scénarios critiques d'une application car coût important => les cibler avec le PO ou les utilisateurs pour mieux les identifier et ne pas perdre du temps à créer des tests pour des scénarios non critiques.
- Si les choses sont bien faites, les scénarios non critiques seront couverts par les tests d'intégration et unitaires.
- Plusieurs outils permettant d'écrire des tests E2E :
 - Selenium (<https://www.seleniumhq.org>) :
 - Fait partie des plus populaires sur le marché.
 - Son utilisation est simple. Par exemple, il suffit de référencer ces packages nuget en `.NET` : `Selenium.Support`, `Selenium.WebDriver`, `Selenium.WebDriver.MicrosoftDriver`
 - Playwright (<https://playwright.dev/dotnet/docs/intro>)
 - Sencha
 - Etc.

Tests exploratoires

- Tests exécutés manuellement.
- Rôle principal : trouver les bugs qui ne sont pas détectés par les autres familles de tests afin d'améliorer la stabilité et l'utilisabilité de l'application.
- Au-delà de la validation, ils permettent aussi l'apprentissage et la compréhension du produit.
- Généralement, ils ne sont pas basés sur des documents et donc non dirigés.
- Leur temps d'exécution peut durer des jours et des jours.

Pour plus de qualité : lancement automatisé des tests

- Une fois les tests créés, on peut les lancer manuellement à chaque modification du code OU automatiser leur exécution en utilisant le live-testing de votre IDE (si l'option existe).
- Le live-testing est très utile lors du développement puisqu'à chaque enregistrement ou compilation, tous les tests présents et concernés par la modification vont être lancés. On voit ainsi si une régression est apportée à condition qu'on ait, bien sûr, une bonne couverture de tests.



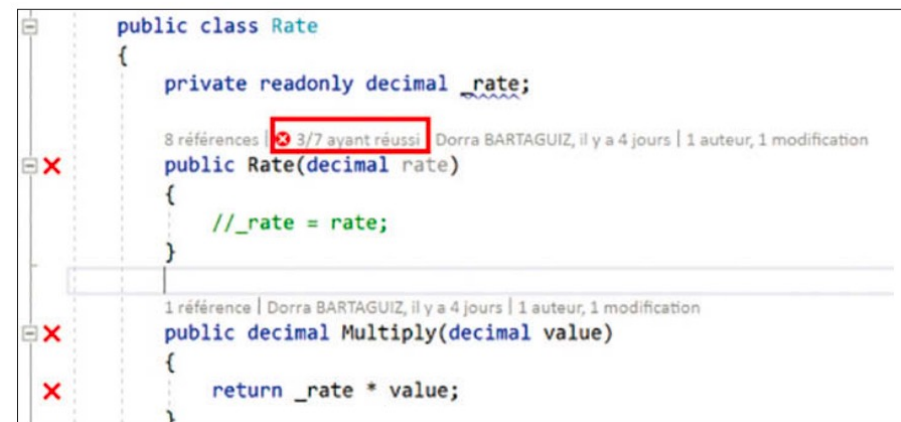
```

public class Rate
{
    private readonly decimal _rate;

    8 références | 7/7 ayant réussi | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification
    public Rate(decimal rate)
    {
        _rate = rate;
    }

    1 référence | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification
    public decimal Multiply(decimal value)
    {
        return _rate * value;
    }
}
  
```

The screenshot shows the source code of the `Rate` class. To the left of the code, there are green checkmarks indicating successful test results. A tooltip for the `Rate` constructor shows "8 références | 7/7 ayant réussi | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification". A tooltip for the `Multiply` method shows "1 référence | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification".



```

public class Rate
{
    private readonly decimal _rate;

    8 références | 3/7 ayant réussi | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification
    public Rate(decimal rate)
    {
        // _rate = rate;
    }

    1 référence | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification
    public decimal Multiply(decimal value)
    {
        return _rate * value;
    }
}
  
```

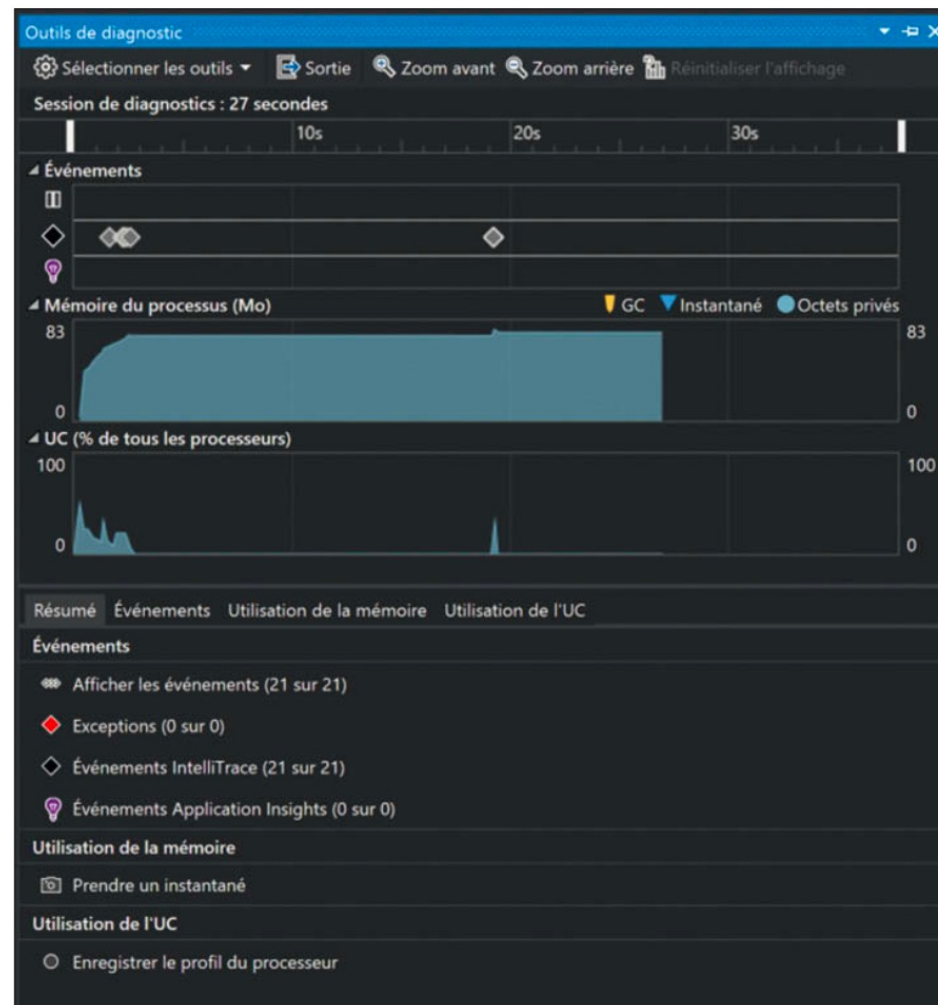
The screenshot shows the source code of the `Rate` class with a modification: the line `_rate = rate;` in the constructor is commented out. To the left of the code, there are red X marks indicating failed test results. A tooltip for the `Rate` constructor shows "8 références | 3/7 ayant réussi | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification". A tooltip for the `Multiply` method shows "1 référence | Dorra BARTAGUIZ, il y a 4 jours | 1 auteur, 1 modification".

Pour plus de qualité : tests de performance

- = Tests de charge ou "stress tests".
- Permettent de s'assurer que l'application supporte bien une certaine charge (de données, d'appels ...) en restant réactive et stable.
- Nécessitent de définir leurs objectifs :
 - Quel est le temps de réponse maximal attendu en s ou ms (temps de réponse serveur et temps d'affichage) ? Le temps de réponse exigé d'une page web ne sera pas forcément le même que celui d'un écran d'un client lourd.
 - Quels sont les modules ou composants qui rentrent dans le scope des tests de performance ?
 - Quel est le nombre d'utilisateurs sur tel ou tel module ou composant ? 2 cas à définir : cas nominal (activité classique) et cas critique (pic d'activités).
- Outils de base :
 - Outil DevTools du navigateur s'il s'agit d'une application web pour calculer le temps de réponse. Ex. : <https://developers.google.com/web/tools/chrome-devtools>
 - Les IDE, comme Visual Studio, intègrent généralement un outil de diagnostic lors du debug de l'application. Il affiche les événements lancés, l'état d'utilisation de la mémoire et du CPU.

Pour plus de qualité : tests de performance

- Ex : outil de diagnostic de VS :



Pour plus de qualité : tests de performance

- Outils avancés :
 - Pour la partie serveur :
 - Apache Bench (<https://httpd.apache.org/docs/2.4/programs/ab.html>)
 - Client :
 - SiteSpeed (<https://www.sitespeed.io/>) : analyse votre site par rapport aux bonnes pratiques en matière de métriques de performance
 - WebPageTest (gratuit, <https://www.webpagetest.org/>) : teste une page sur le navigateur choisi et remonte des informations précises sur les performances de la page.

Pour plus de qualité : tests de vérification en service régulier

- Une fois que l'application est livrée en production, il s'agit de vérifier si elle est fonctionnelle.
- La vérification peut se faire :
 - Manuellement en lançant, par exemple, tous les matins ou x temps, l'application
 - Ou être automatisée :
 - Exemple : Utiliser des tests End To End, par exemple, pour voir si la page se charge bien.

Pour aller plus loin

- Conférence de Nicolas Fédou (Coach Craft, coding architect, formateur, tech lead, développeur Java chez Arolla) au Flowcon'18 : <https://www.infoq.com/fr/presentations/flowcon-2018-Nicolas-Fedou/>
- Remarques :
 - Valeurs Craft :
 - Vous avez soigné votre code en le refactorant après avoir appliqué un correctif
 - En ajoutant un test automatisé, vous avez garanti la non-régression pour éviter de corriger toujours les mêmes bugs. Ainsi vous pourrez continuer d'ajouter de la valeur.
 - En documentant votre code et en le montrant à un autre développeur, vous avez favorisé l'appropriation collective du code
 - Vous avez établi une relation de partenariat avec votre client en prenant la responsabilité de soigner l'aspect technique (tests unitaires, refactoring) au delà d'un correctif minimal
 - https://fr.wikipedia.org/wiki/Software_craftsmanship
 - <http://manifesto.softwarecraftsmanship.org/#/fr-fr>



Quand mon collègue pousse du code
en prod sans avoir vraiment testé

