TDD, refactoring et code propre

R2.03 - Qualité de développement

IUT d'Aix-Marseille - BUT Informatique, première année

TDD

Kata

Refactoring

Au programme de ce CM

Trois axes pour transformer les bases du CM1 en **réflexes** : approfondir le TDD, pratiquer ensemble, sécuriser du legacy.

TDD approfondi

F.I.R.S.T., stratégies de Beck, ApprovalTests

Kata et pair programming

Driver, navigator, ping-pong

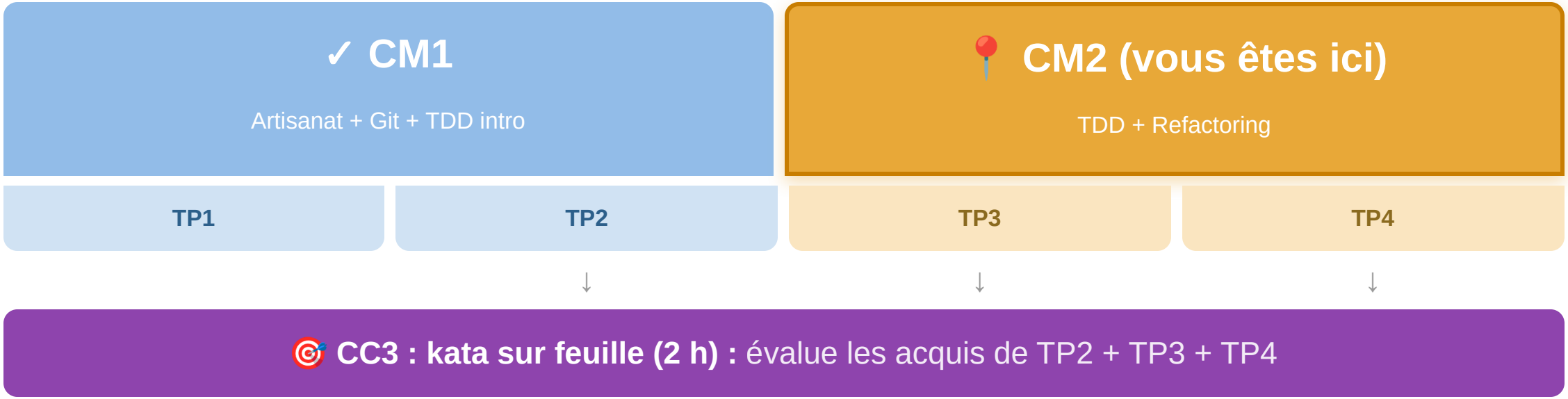
Code smells et refactoring

Fowler, characterization tests

Ce CM prépare le **TP3 - Kata** et le **TP4 - Refactoring**.

Où on en est

Aujourd'hui, on **approfondit** le socle du CM1 et on prépare les **TP3**, **TP4** et la logique du **CC3**.



En CM1 on a vu *pourquoi* l'artisan prend soin de son code. Aujourd'hui, on voit **comment il s'y prend au quotidien** : tester avant, refactorer souvent, nommer juste. Les gestes qui transforment l'intention en habitude.

Partie 1 - TDD approfondi

Comment tirer le meilleur du cycle du TDD au quotidien ?

🔁 Rappel : le cycle RED-GREEN-REFACTOR

Pour rappel : **3 étapes**, dans cet ordre, pour chaque petit comportement à ajouter.

1. 🔴 RED

J'écris **un seul** test. Il doit échouer pour la **bonne raison** : un comportement qui manque, pas une coquille de syntaxe.

▶

2. 🟢 GREEN

Je code le **minimum** pour passer. Je n'ai **pas** le droit d'anticiper sur les tests à venir.

▶

3. 🧹 REFACTOR

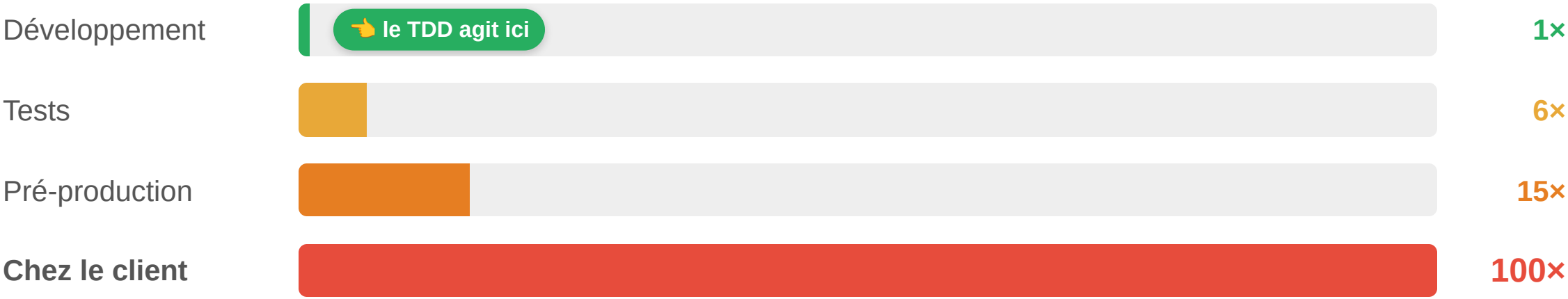
Je nettoie (noms, duplication, lisibilité). La suite de tests reste **verte** en permanence.

🔁 ... puis on **reprend en 1.** avec le test suivant.

La force du cycle vient de sa **discipline** : 3 étapes, dans l'ordre, à chaque tour.

💰 Pourquoi ce cycle, économiquement

Le CM1 a montré que les bugs coûtent jusqu'à **100×** plus cher quand ils sont détectés tard. Le TDD attaque la courbe **au moment le moins cher**.



Le TDD ne supprime pas le travail ; il le **déplace là où il coûte le moins cher**.

🚦 Les 3 lois du TDD (Uncle Bob)

Trois règles formelles, énoncées par **Robert C. Martin** (Uncle Bob), pour **forcer la discipline du baby-step**.

🙅 Loi 1 : Interdit

Tu n'écriras **pas de code de production** tant qu'un test qui **échoue** ne l'exige.

⏸️ Loi 2 : Limite

Tu n'écriras **pas plus de test** qu'il n'en faut pour échouer. Une erreur de compilation compte comme un échec.

✋ Loi 3 : Juste assez

Tu n'écriras **pas plus de code de production** qu'il n'en faut pour faire passer le test qui échoue.

À chaque cycle, vous alternez entre production et test toutes les **30 secondes à 2 minutes**.

🎯 Fake it / Triangulation / Obvious : laquelle choisir ?

On les a vues au CM1. Voici la **décision rapide** au moment d'écrire le code de production.

🎭 Fake it d'abord, toujours

Même si la solution semble évidente. Fake it vérifie que le **test** est juste, **avant** de se soucier du code.

💡 *Le test passe avec une constante en dur ? OK, il est bien formulé.*

📐 Triangulation quand...

... le 2e ou 3e test vous **force** à généraliser. Le comportement commun émerge *tout seul*.

💡 *FizzBuzz : après 3 tests (1, 3, 5), la logique if/else se dessine.*

💡 Obvious seulement si...

... la solution **tient en une ligne** et vous ne pouvez *pas* vous tromper. **Le moindre doute** = pas obvious.

💡 `return a + b;` *dans* `addition(int, int)`.

Règle simple : **Fake it** par défaut, **Triangulation** quand un seul cas ne suffit plus, **Obvious** seulement quand l'évidence est totale.

🎯 Fake it / Triangulation / Obvious : laquelle choisir ?

On les a vues au CM1. Voici la **décision rapide** au moment d'écrire le code de production.

🎭 Fake it d'abord, toujours

Même si la solution semble évidente. Fake it vérifie que le **test** est juste, **avant** de se soucier du code.

💡 *Le test passe avec une constante en dur ? OK, il est bien formulé.*

📐 Triangulation quand...

... le 2e ou 3e test vous **force** à généraliser. Le comportement commun émerge *tout seul*.

💡 *FizzBuzz : après 3 tests (1, 3, 5), la logique if/else se dessine.*

💡 Obvious seulement si...

... la solution **tient en une ligne** et vous ne pouvez *pas* vous tromper. **Le moindre doute** = pas obvious.

💡 `return a + b;` dans `addition(int, int)`.

⚠️ **Anti-pattern fréquent** : « je sais coder ça, je passe direct en obvious ». Le test n'est jamais rouge, vous ne savez pas s'il vérifie vraiment quelque chose. **Fake it d'abord**, même 10 secondes, pour voir le test passer du rouge au vert.

Règle simple : **Fake it** par défaut, **Triangulation** quand un seul cas ne suffit plus, **Obvious** seulement quand l'évidence est totale.

✅ Qu'est-ce qu'un bon test ? Le principe F.I.R.S.T.

Robert C. Martin (*Clean Code*, 2008) a énoncé **5 critères** d'un bon test, l'acronyme **FIRST**.

Fast	**I**ndependent	**R**epeatable	**S**elf-validating	**T**imely
Rapide (quelques ms). On les lance des dizaines de fois par heure.	**Indépendants** entre eux. Changer l'ordre ne doit rien casser.	**Reproductible**. Pas de dépendance au réseau, à l'heure, à un fichier temporaire.	**Auto-vérifiant**. Rouge ou vert, pas de « regardez le log ».	Écrit au **bon moment** : **avant** le code, pas après.

🛠️ F.I.R.S.T., c'est la **check-list de l'artisan** qui vérifie que son banc de tests reste un banc, pas un gouffre. Un test lent, fragile, dépendant des voisins, c'est un outil émoussé qu'il faut **affûter ou jeter**.

🧪 La structure AAA d'un test

Trois phases pour structurer chaque test : **Arrange · Act · Assert**. Lisible comme une phrase.

🛠️ **Arrange** : préparer l'environnement, les données, les dépendances

▶️ **Act** : déclencher le comportement testé (**une seule** action)

✅ **Assert** : vérifier le résultat (idéalement **une** assertion principale)

📄 `CalculatriceTest.java`

```java
@Test
void additionne_deux_entiers_positifs() {
    // Arrange
    Calculatrice c = new Calculatrice();
    // Act
    int resultat = c.additionner(2, 3);
    // Assert
    assertThat(resultat).isEqualTo(5);
}
```

Un test **bien structuré** (AAA) et **bien nommé** se lit comme une phrase.

📝 Exemple : noms de tests parlants

Le nom du test est sa **première ligne de documentation**. Il doit raconter le **quoi** du test et **quel comportement** est attendu.

🤦 Pas clair

```
@Test void test1()
@Test void testAdd()
@Test void testCalc()
@Test void bugFix12345()
```

On ne sait **pas ce qui est testé**, ni **quel comportement** est attendu.

🎯 Clair

```
@Test void additionne_deux_positifs_retourne_un_positif()
@Test void additionne_zero_et_dix_retourne_dix()
@Test void montant_total_d_un_panier_negatif_leve_exception()
@Test void un_panier_vide_a_un_montant_total_de_zero()
```

On lit le test comme une **spécification** du comportement de chaque cas.

🥋 **Ce qu'on vise au TP3** : des tests qui racontent une partie, pas des numéros ou des noms génériques : `partie_tombe_a_egalite_apres_deux_points_partout()`, `le_serveur_gagne_la_partie_apres_quatre_points()`, etc.

Conventions de nommage

Le **snake_case** est la norme JUnit pour la lisibilité. Trois **patterns** structurent le contenu du nom : choisissez-en un et tenez-vous-y.

📝 Descriptif simple

Décrit le comportement **sans structure imposée**. Le plus naturel à lire.

```
additionne_deux_positifs_retourne_la_somme()
```

should / when

Pattern **should_X_when_Y**. Très répandu en Java/Kotlin.

```
should_return_sum_when_adding_two_positives()
```

🎬 Given / When / Then

Pattern BDD : **given_X_when_Y_then_Z**. Aligne le test sur le scénario métier.

```
given_two_positives_when_adding_then_returns_sum()
```

Pas de meilleur pattern : la **cohérence dans le projet** prime sur le style.

🥋 Grounding : ces principes sur un vrai kata

Le kata **Tennis** du TP3 illustre tout ce qu'on vient de voir. Voici les 3 premiers tests pris dans **leur ordre d'écriture**.

🎭 Test 1 : Fake it

```
@Test
void debut_de_partie_est_0_0() {
  assertThat(new Tennis().score())
    .isEqualTo("0-0");
}

// Impl : return "0-0";
```

Le test passe avec une constante. On sait que le test est **juste**.

📐 Test 2 : Triangulation

```
@Test
void apres_point_serveur_15_0() {
  Tennis t = new Tennis();
  t.pointPourServeur();
  assertThat(t.score())
    .isEqualTo("15-0");
}
```

Force à stocker un **état**. La constante ne suffit plus.

📐 Test 3 : Triangulation

```
@Test
void apres_2_points_serveur_30_0() {
  Tennis t = new Tennis();
  t.pointPourServeur();
  t.pointPourServeur();
  assertThat(t.score())
    .isEqualTo("30-0");
}
```

Force une vraie **table** (0, 15, 30, 40). La logique émerge.

Chaque test est **un pas**. La conception **s'invente au fur et à mesure**, pas avant.

🎭 La doublure de test : à quoi ça sert ?

Au cinéma, le héros a une **doublure** pour les cascades. En test, on fait pareil avec les dépendances qu'on ne contrôle pas (base de données, réseau, horloge) : on les remplace par une **doublure de test** (anglais : *test double*).

🪨 Stub / Fake

Forcer les valeurs retournées par la dépendance. On contrôle la valeur qu'elle nous *renvoie* pendant le test.

Exemple : une horloge qui retourne toujours `1er janvier 2026`.

🎯 Mock / Spy

Observer l'appel. On vérifie *comment* la dépendance a été utilisée (méthode, arguments, fréquence).

Exemple : vérifier qu'un service mail a bien été appelé une fois avec la bonne adresse.

Isoler ce qu'on ne contrôle pas, pour **tester** ce que l'on souhaite maitriser.

🪨 Le stub : forcer le retour

Un **stub** remplace une dépendance par une **réponse pré-définie**. Vous décidez ce que la dépendance retourne et vos tests deviennent **reproductibles**.

📄 Exemple : une horloge stub

```
// Production : dépend de l'horloge système
Clock horlogeStub = Clock.fixed(
    Instant.parse("2026-01-01T00:00:00Z"),
    ZoneOffset.UTC);

LocalDate date = LocalDate.now(horlogeStub);
// → toujours 2026-01-01, peu importe le jour réel
```

💡 Quand l'utiliser

- Tester un code qui dépend du **temps**, du **hasard**, d'un **fichier**
- Rendre les tests **reproductibles** (même résultat à chaque exécution)
- Éviter d'appeler une **vraie** base de données ou un **vrai** service externe

Un stub **FORCE le retour** de la dépendance. On ne vérifie **pas** comment elle a été appelée.

🎯 Le mock : vérifier l'appel

Un **mock** vérifie **comment** une dépendance est utilisée : *quelle méthode*, *avec quels arguments*, *combien de fois*.

📄 Exemple : un service mail mock (Mockito)

```
ServiceMail mail = mock(ServiceMail.class);
GestionnaireCommande gc
    = new GestionnaireCommande(mail);

gc.confirmer(new Commande("alice@iut.fr"));

// On VÉRIFIE l'appel
verify(mail).envoyer(eq("alice@iut.fr"),
    contains("confirmation"));
```

💡 Quand l'utiliser

- Vérifier qu'une **action** a bien eu lieu (envoi, log, notification)
- Tester une **interaction** entre deux composants
- S'assurer du **contrat** entre votre code et la dépendance

Un mock **VÉRIFIE l'appel**. La valeur de retour est secondaire.

🎭 Stub vs Mock : la synthèse

Quatre aspects pour distinguer les deux doublures, dans un seul coup d'œil.

Aspect	🪨 Stub	🎯 Mock
But	Forcer le **retour**	Vérifier **l'appel**
Question	*Que renvoie X ?*	*Comment X est appelé ?*
Assertion porte sur	Le **résultat**	L'**interaction**
Exemple typique	Horloge fixe, base de données en mémoire	Service mail, logger

Vous voulez **vérifier une interaction** → c'est un **mock**. Sinon → c'est un **stub**.

📸 ApprovalTests : quand la sortie est textuelle

Pour du code qui produit une **sortie complexe** (grille, JSON, rapport), écrire l'attendu à la main est pénible. **ApprovalTests** automatise le rituel.

📄 **GrilleDemineurTest.java**

```
@Test
void grille_demineur_5x5() {
  List<String> entree = List.of(
    " *   ",
    "     ",
    "  *  ",
    "     ",
    "   * "
  );
  List<String> sortie =
    new GrilleDemineur(entree).annotee();
  Approvals.verifyAll("", sortie);
}
```

🔁 Le rituel d'approbation

1. Je lance le test
2. La sortie va dans `...received.txt`
3. Je **vérifie visuellement**
4. Je renomme `received` → `approved`
5. Le test compare ensuite `received` à `approved`

Utilisé au **TP2 exercice 5** (Démineur) : gain énorme sur les grandes grilles ou les sorties textuelles riches.

💡 Règles pour ne pas se planter en TDD

Trois pièges à éviter, trois réflexes à cultiver. Les premiers tuent le filet de sécurité, les seconds le renforcent.

🤦 Pièges à éviter

- **Écrire plusieurs tests d'affilée** sans coder entre. Un test écrit, un test codé.
- **Ajouter du code sans test qui l'exige.** Une branche `if` en plus = un test à écrire avant.
- **Sauter le REFACTOR.** Le cycle est à 3 étapes, pas à 2. Sinon la dette explose.

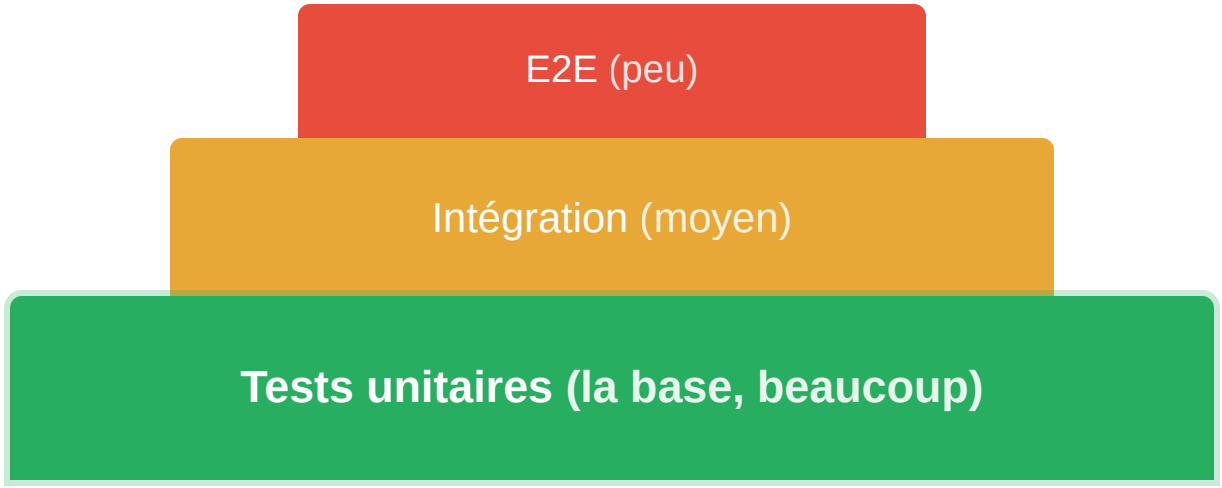
🎯 Réflexes à cultiver

- **Vérifier que le test échoue** avant de coder. Sinon = faux positif (un test qui passait déjà).
- **Commit à chaque GREEN ou REFACTOR.** Chaque état vert = un point de sauvegarde.
- **Un seul test rouge à la fois.** Focaliser l'attention, pas 5 rouges en parallèle.

Le TDD ne pardonne pas les **raccourcis** : chaque écart efface une garantie acquise.

📊 Pyramide des tests

Pour rappel (CM1) : **3 niveaux** de tests, en pyramide. Plus on monte, moins on en a, et plus chaque test coûte cher.



E2E : toute l'application, du clic à la base. Lents, fragiles, coûteux.

Intégration : plusieurs briques ensemble (service + DB, par ex).

Unitaires : une méthode, une classe, isolés. Rapides, ciblés, relancés en permanence.

👉 Dans ce module, on travaille la **base** de la pyramide. C'est là que le TDD vit vraiment.

🧬 Aparté : couverture != qualité des tests

Deux mesures qu'on confond souvent. La **couverture** répond à *quelles lignes ont été exécutées ?* La **qualité** répond à *les tests détectent-ils de vrais bugs ?*

📊 Couverture

Quelles **lignes / branches** ont été exécutées par la suite de tests.

Mesurée par **JaCoCo** (plugin Maven), inline IntelliJ, etc.

🎯 Qualité

Les tests **détectent-ils vraiment un bug** si le code est cassé ?

Mesurée par **mutation testing** (CM2 bonus) ou par **lecture critique** (pendant la revue de code).

La couverture mesure **l'exécution**, pas la **validation**. Un beau 99% ne prouve rien à lui seul.

🌄 Où s'arrête le TDD ?

Connaître les **limites** de l'outil, c'est déjà bien le maîtriser. Le TDD résout certains problèmes très bien mais il en laisse d'autres entiers.

🎯 Ce que le TDD fait bien

- Cadrer une **intention** avant de coder
- Donner un **filet** pour refactorer sereinement
- Produire une **documentation vivante** du comportement
- Forcer un **découplage** (sinon le test devient injouable)

🚧 Ce que le TDD ne fait PAS

- **Garantir** que le produit est utile pour le client
- Remplacer la **revue de code** et le **pair programming**
- Couvrir les bugs d'**intégration**, de **perf**, d'**ergonomie**
- Dispenser de **penser** : un test tautologique passera toujours

L'artisan connaît ses outils et leurs **limites**. Le TDD est un outil puissant pour **le code qui porte une logique claire** ; il ne remplace ni les tests d'intégration, ni la relecture humaine, ni la validation par le client qui dira si l'ouvrage correspond bien à ses exigences.

Partie 2 - Le kata et le pair programming

Comment on devient meilleur ? En répétant des gestes simples.

🥋 Le coding dojo : pratiquer pour apprendre

Comment apprend-on à mieux coder ? Pas en regardant des tutos, ni en lisant des livres seul. Comme dans les arts martiaux : par la **pratique régulière, répétée et en groupe**.

> 💬 *« Si je veux apprendre le judo, je vais m'inscrire au dojo du coin et y passer une heure par semaine pendant deux ans, au bout de quoi j'aurai peut-être envie de pratiquer plus assidûment. Si je veux apprendre la programmation objet, mon employeur va me trouver une formation de trois jours à Java dans le catalogue. Cherchez l'erreur. »*
>
> **Laurent Bossavit**, *The Leprechauns of Software Engineering*, 2013

🥋 Le coding dojo

- **Origine** : transposition du dojo des arts martiaux à la programmation. Premiers coding dojos vers 2005 (Paris, Londres).
- **Forme typique** : 1 à 2 h, en groupe, sur un kata, avec un facilitateur.
- **Fréquence** : régulière (hebdo / mensuelle). La régularité prime sur la durée.

Le métier s'apprend par la **pratique régulière**, pas par une formation one-shot.

🥋 Qu'est-ce qu'un kata ?

Dans le coding dojo, l'unité de pratique s'appelle un **kata**. Le mot vient directement des arts martiaux.

🥋 Arts martiaux

Une **séquence de mouvements** qu'on répète jusqu'à ce que le corps l'exécute sans y penser.

On ne cherche pas à *vaincre un adversaire* : on travaille la **posture**, le **souffle**, la **précision**.

💻 Coding kata

Un **petit exercice** qu'on refait **plusieurs fois** pour :

- **automatiser** des gestes (raccourcis IDE, TDD, refactoring)
- **essayer** des approches différentes
- **comparer** avec d'autres développeurs

Popularisé par **Dave Thomas** (*The Pragmatic Programmer*, 2003).

Le but n'est pas de **résoudre le problème** (vous le connaissez), c'est d'améliorer **votre façon** de le résoudre. Comme un pianiste qui joue ses gammes pour **garder la main**.

🥋 Quelques kata célèbres

La communauté des coding dojos a accumulé un **répertoire** de kata classiques. En voici six qu'on retrouve partout, chacun éclaire un aspect différent du métier.

🔢 FizzBuzz

Le plus **iconique** : intro TDD, branchements simples. La porte d'entrée.

🗓️ Années bissextiles

Petit kata idéal pour travailler les **règles booléennes** imbriquées.

🎾 Tennis scoring

Afficher le score d'un jeu de tennis. Idéal pour les **state machines**.

🎳 Bowling

Kata classique d'**Uncle Bob**. Marquer une partie de bowling avec strikes et spares.

🎲 Yahtzee

Scorer les combinaisons d'un jet de 5 dés. Bon terrain pour la **stratégie**.

🏪 Gilded Rose

Par **Emily Bache**. Kata de **refactoring** sur du legacy volontairement horrible.

Au **TP3**, vous en ferez **cinq** par vous-même : Années bissextiles, Tennis, Gestion employés, Pagination, Yahtzee.

🤔 Pourquoi refaire un kata qu'on sait résoudre ?

Le débutant croit que refaire = perdre du temps. L'expert sait que **chaque passage** apporte autre chose. Voici ce qui change au fil des répétitions.

1re fois : la lutte

Je galère, je découvre les règles. Je code *quelque chose qui marche*.

2e fois : l'aisance

Je vais plus vite, je vois les **pièges**. Je soigne le **nommage**.

3e fois : l'aventure

Je teste une **autre approche** : que du Fake it, ou que de la Triangulation.

10e fois : la maîtrise

Je suis **fluide**. Je peux me concentrer sur le **style**, pas le problème.

Un sportif ne s'entraîne pas le jour du match. **Un développeur non plus.**

🎲 Varier les contraintes pour progresser

Refaire un kata avec une **règle du jeu différente** force à explorer d'autres chemins.

🙅 Pas de `if` / `else`

Force le **polymorphisme** et les *lookup tables*. Apprend à remplacer les branches par des objets.

🔁 Pas de boucles

Force la **récursion** ou les `Stream`. Apprend la programmation fonctionnelle.

📦 Pas de primitives

Force à créer de vrais **objets métier**. Apprend *Tell, Don't Ask*.

✂️ 4 lignes max

Méthodes courtes obligatoires. Apprend **Extract Method** à fond et la décomposition fine.

🤐 Silence

En pair, communiquer **uniquement** par le code et les tests. Apprend à l'expressivité.

🏓 Mute ping-pong

A écrit le test, B le fait passer, **sans parler**. Apprend la rigueur TDD pure.

Une **contrainte arbitraire** transforme un kata connu en **nouveau terrain d'exploration**.

👥 Le pair programming

Deux personnes, un clavier. Pratique née de l'**eXtreme Programming** (Kent Beck, 1996) : **chaque ligne** de code est écrite à deux.

🎮 Driver

Celui qui tape. Se concentre sur le **comment** : syntaxe, IDE, faire passer le test courant.

🧭 Navigator

Celui qui pense. Se concentre sur le **quoi** : design, cas limites, lisibilité, prochaine étape.

✅ Bénéfices

Moins de bugs, revue de code en continu, montée en compétence croisée, *bus factor* réduit, effet antisomnolent 😴.

⚠️ Pièges

Ce n'est **pas** "un qui code, un qui regarde son téléphone". Fatigant : **pauses** toutes les heures et rotation toutes les **7 à 10 min**.

Le pair, ce n'est pas un duo regardeur / codeur. C'est **deux esprits qui pensent ensemble** au même problème.

🏓 Variante du pair programming : le ping-pong

Variante **ludique** du pair programming : on combine pair et **cycle TDD strict**. Chacun à son tour **défie** l'autre avec un test ; l'autre doit le faire **passer**. Puis on inverse.

🏓 Manche 1 : 🅰️ sert

🅰️ écrit un **test** qui échoue 🔴

⬇

🅱️ fait **passer** le test 🟢

🏓 Manche 2 : 🅱️ sert

🅱️ écrit un **nouveau test** qui échoue 🔴

⬇

🅰️ fait **passer** le test 🟢

À chaque manche, **les rôles s'inversent** : celui qui posait la question doit maintenant y répondre.
Entre les manches : **refactor à deux**.

🏓 Pourquoi faire du ping-pong ?

Le jeu n'est pas gratuit : cette alternance forcée produit **trois bénéfices** impossibles à obtenir en solo.

🎨 Design partagé

Chacun **devine l'intention** de l'autre. Le design émerge à **deux têtes**, pas une seule.

🛡️ Discipline imposée

Impossible d'écrire du code *en douce* sans test : le coéquipier **veille** au cycle.

🔍 Tests lisibles

Un test mal écrit se voit **tout de suite** : l'autre ne sait pas quoi coder.

On ne peut pas tricher : le test et le code sont écrits par **deux personnes différentes**.

👥 Le mob programming

Quand le pair programming s'étend à toute l'équipe : un seul écran, **tout le monde sur le même problème**, en même temps. Inventé par **Woody Zuill** chez Hunter Industries (~2014).

⚙️ Le principe

- **1 driver** aux mains du clavier
- **N navigators** qui dirigent à voix haute
- **Rotation** du driver toutes les 5 à 10 min
- Tout le monde dans la **même pièce** (ou même Zoom)

🎯 À quoi ça sert

- Problèmes **complexes** à découper en équipe
- Démarrage d'un **nouveau projet** ou d'un module critique
- **Onboarding** rapide d'un nouvel arrivant
- Décisions d'**architecture** partagées par tous

À 2 = **pair**, à 3+ = **mob**. Plus on est, plus la décision se construit en commun, au prix du débit individuel.

🏙️ La pratique en entreprise

Ces pratiques (kata, pair, mob) ne sont pas réservées à l'école. Elles sont **utilisées au quotidien** dans les équipes qui prennent au sérieux la qualité du code.

🌍 Où on les trouve

- Équipes de **craftsmanship** : Pivotal Labs, ThoughtWorks, ekino, Octo, Zenika...
- Projets **open source** de référence (Linux kernel, Mozilla)
- Entreprises avec culture **XP / agilité forte**
- Hackathons et conférences (Devoxx, AgileFrance, BDX I/O...)

🎯 Pourquoi c'est viable

- **Moins de bugs** : la revue est intégrée à l'écriture
- **Montée en compétence cross-team** : tout le monde apprend de tout le monde
- **Onboarding** beaucoup plus rapide
- Réduction du **bus factor** (personne n'est seul à connaître un module)

Apprendre par la pratique régulière, en groupe, ce n'est pas un luxe d'école mais de la **formation continue**. C'est comme ça que **les meilleures équipes** restent **les meilleures**.

🎯 Démo en ouverture du TP3

Au début du TP3, on fera **20 minutes de kata live** en **mob programming** : tout le groupe sur le même problème, l'enseignant au clavier. Objectif : voir **à quoi ressemble** une séance TDD avant de passer en binôme sur les exercices.

🥋 Le kata retenu

Tennis scoring, un classique du dojo :

- **États** : 0, 15, 30, 40, deuce, advantage, win
- **Transition** : marquer un point pour A ou B
- **Sortie** textuelle : *"Player 1 : 30 / Player 2 : 15"*
- **Pièges** : égalité à 40-40, regagner l'avantage

👀 Ce que vous verrez

- L'enseignant reste **driver** (au clavier)
- Le **navigator** tourne dans le groupe toutes les **2 min** au chronomètre
- Comment une solution se construit par **consensus** du groupe
- **Commit** à chaque fin de **cycle TDD** (après le refactor)

Après la démo, c'est **à vous** : 6 kata en binôme, vous tournez les rôles toutes les 5 à 10 minutes.

Partie 3 - Code smells et refactoring

Comment vivre avec du code qui marche, mais nous fait honte ?

🛠️ Refactorer, c'est entretenir ses outils

L'artisan ne nettoie pas ses outils par perfectionnisme. Il le fait parce que demain, **il s'en sert encore**. Remanier son code, c'est exactement ce même geste et ce même besoin mais pour un développeur qui maitrise sa dette technique.

🪚 Le menuisier

Aiguise sa scie **entre deux chantiers**. Une lame émoussée fait perdre du temps à chaque coupe et elle peut blesser.

🍳 Le chef

Récure et aiguise ses couteaux **en fin de service**. Demain, on ne commence pas avec une cuisine sale et des couteaux émoussés.

💻 Le développeur

Remanie son code **après chaque GREEN**. Pas pour faire joli mais pour **rester rapide** sur la fonctionnalité suivante tout en conservant la confiance en son code.

Le code qu'on lit cette semaine, c'est celui qu'on **modifiera** le mois prochain. Le garder propre à chaque instant garantit qu'on pourra compter sur lui quand on en aura besoin.

👃 Qu'est-ce qu'un code smell ?

Terme inventé par **Kent Beck** et popularisé par **Martin Fowler** dans *Refactoring* (1999, ~20 smells catalogués). Un code smell est un **indice** que quelque chose ne va pas, pas un bug ni une erreur de compilation.

🚨 Un signal, pas une alarme

Le code **compile**, les tests **passent**, l'utilisateur ne voit rien. Mais quand on essaie de le **modifier**, ça résiste : la zone va devenir pénible pour les développeurs.

🍞 Comme un aliment qui sent

Pas forcément périmé, mais il faut **regarder de près**. Le smell est un signal d'enquête, pas une condamnation.

📏 Un vocabulaire partagé

Long Method, *Magic Number*, *Duplicated Code*... un **nom commun** qui rend les revues de code plus rapides : pas besoin de réexpliquer le problème à chaque fois.

"If it stinks, change it." - **Kent Beck**

📚 Les smells qu'on va surtout rencontrer au TP4

Fowler en référence une vingtaine. Voici les **6 smells** que vous croiserez au TP4, chacun avec son refactoring associé.

📏 Long Method

Une méthode de 50 lignes qu'on doit scroller pour la lire.

→ **Extract Method**

🏢 Large Class

Une classe qui fait **trop de choses** à la fois.

→ **Extract Class**

📋 Long Parameter List

Une méthode à **7 paramètres**, imbuvable à l'appel.

→ **Introduce Parameter Object**

👯 Duplicated Code

Le même bloc **copié-collé** en 3 endroits.

→ **Extract Method / Move Method**

🔢 Magic Numbers

`if (age > 65)`, `total *= 1.20`.

→ **Replace Magic Number with Constant**

🔀 Switch Statements

Un `switch` sur un type, **dupliqué** en 5 endroits.

→ **Replace Conditional with Polymorphism**

Un smell n'est pas une **faute**, c'est un **signal** qu'une zone va devenir pénible à modifier.

🔧 Qu'est-ce qu'un refactoring ?

Le mot *refactoring* (en français : **remaniement**) est utilisé à toutes les sauces. La définition est plus **exigeante** qu'on ne le croit :

> 💬 *"Modifier la **structure interne** du code **sans changer son comportement observable**."*
>
> - **Martin Fowler**, *Refactoring: Improving the Design of Existing Code* (1999, 2018)

✅ C'est un refactoring	🙅 Pas un refactoring
Renommer une variable, extraire une méthode, déplacer un champ dans une autre classe. → **Les tests existants continuent de passer.**	Changer ce que fait une méthode, ajouter une fonctionnalité, corriger un bug. → **Là, les tests changent (ou deviennent rouges).**

🛡️ Prérequis absolu : une **suite de tests** qui couvre le comportement. Sinon, c'est de la **réécriture** (*rewriting*), pas un **remaniement** (*refactoring*).

📖 Le catalogue de Fowler (70+ refactorings)

Fowler a normalisé ce vocabulaire : **70+ transformations** cataloguées, chacune avec son **nom**, son **contexte d'usage** et sa **procédure pas-à-pas**. Tout est en ligne sur refactoring.com/catalog.

🏷️ Rename

Le plus fréquent. Pour un **nom plus juste**, qui révèle l'intention.

✂️ Extract Method

Découper une **longue méthode** en plus petites, chacune avec une intention claire.

📦 Extract Class

Sortir un **sous-ensemble cohérent** de la classe trop chargée dans sa propre classe.

🦎 Polymorphism

Remplacer un `switch` sur un type par des **sous-classes**.

Au TP4, vous pratiquerez ces **4 grands classiques** avec l'aide de votre IDE.

🧪 Exemple : Extract Method

On découpe une longue méthode en plusieurs petites, chacune nommée d'après son **intention**. Le code racine devient un **sommaire**.

🦴 Avant : Long Method

```
void imprimerFacture(Client c, List<Article> a) {
  // entête
  System.out.println("=== FACTURE ===");
  System.out.println("Client : " + c.nom());
  System.out.println("Date : " + LocalDate.now());

  // lignes
  for (Article art : a) {
    System.out.printf("  %s x%d : %.2f€%n",
      art.nom(), art.qte(), art.prix() * art.qte());
  }

  // total
  double t = 0;
  for (Article art : a) t += art.prix() * art.qte();
  System.out.printf("TOTAL : %.2f€%n", t * 1.20);
}
```

✨ Après : intentions nommées

```
void imprimerFacture(Client c, List<Article> a) {
  imprimerEntete(c);
  imprimerLignes(a);
  imprimerTotal(a);
}

void imprimerEntete(Client c) {
  System.out.println("=== FACTURE ===");
  System.out.println("Client : " + c.nom());
  System.out.println("Date : " + LocalDate.now());
}

void imprimerLignes(List<Article> a) { /* ... */ }

double calculerTotalHT(List<Article> a) {
  return a.stream()
    .mapToDouble(x -> x.prix() * x.qte()).sum();
}
```

🧪 Exemple : Replace Conditional with Polymorphism

On remplace un `switch` sur un **type** par une **hiérarchie de sous-classes**. Chaque branche devient une méthode dans sa sous-classe.

🦴 Avant : Switch Statements

```
class Animal {
  String type;
  String faireDuBruit() {
    switch (type) {
      case "chien":  return "Wouf !";
      case "chat":   return "Miaou !";
      case "vache":  return "Meuh !";
      case "canard": return "Coin coin !";
      default: throw new IllegalStateException();
    }
  }
}
```

Chaque nouveau type oblige à **rouvrir** cette classe. Le `switch` est souvent **dupliqué** ailleurs (parler, manger, dormir...).

✨ Après : Polymorphisme

```
abstract class Animal {
  abstract String faireDuBruit();
}
class Chien extends Animal {
  String faireDuBruit() { return "Wouf !"; }
}
class Chat extends Animal {
  String faireDuBruit() { return "Miaou !"; }
}
class Vache extends Animal {
  String faireDuBruit() { return "Meuh !"; }
}
```

Nouveau type = **nouvelle classe**. On ne touche **pas à l'existant**.

Principe **Open/Closed** : ouvert à l'**extension** (ajouter une classe), fermé à la **modification** (ne pas toucher l'existant).

🛠️ Votre IDE fait le gros du travail

Un refactoring **manuel** est risqué : un oubli, un remplacement raté → le comportement change. L'IDE fait le travail à votre place, **sans erreur**. Au TP4, on abuse de deux raccourcis dans le **Codespace VS Code** (extension Java Red Hat).

⌨️ `Ctrl+.` Quick Fix + Refactor

Menu **contextuel** selon ce que touche le curseur :

- *Extract to method / constant / variable*
- *Move*, *Inline*
- *Add unimplemented methods*

🏷️ `F2` Rename

Renomme **partout** en un coup : déclaration, usages, imports, Javadoc.

Le refactoring le plus **fréquent**, et celui pour lequel l'IDE est **indispensable**.

💡 Si l'IDE sait faire le refactoring : **laissez-le** faire. Si vous devez le faire à la main : **ajoutez d'abord des nouveaux tests**.

IntelliJ IDEA propose les mêmes refactorings avec `Ctrl+Alt+M`, `Ctrl+Alt+V`, `Shift+F6`...

🛡️ Characterization tests : sécuriser du legacy

Concept de **Michael Feathers** (*Working Effectively with Legacy Code*, 2004). **Legacy** = code **sans tests** qu'on doit faire évoluer. Sans filet, pas de refactoring en sécurité.

1. 🔍 Observer	2. ✍️ Figer	3. ✅ Verrouiller	4. 🔧 Refactorer
Lancer le code sur des entrées variées, **noter** ce qu'il sort.	Écrire un test qui **attend exactement cette sortie**, juste ou pas.	Le test passe : le comportement actuel est **fixé**, vous avez un filet.	Si un test casse : vous avez **changé le comportement**, donc créé un bug.

Ces tests ne valident pas que le code est **juste**. Ils le **pinnent**. C'est à dire qu'ils figent ce qui sort, pour que le refactoring n'introduise pas de régression.

🛡️ Exemple : un code opaque à caractériser

Une méthode `fraisPort` empile **5 règles** qui s'additionnent, s'écrasent, se multiplient. Impossible de prédire le résultat *juste en lisant*.

🦴 Le code legacy (en production, pas un seul test)

```
double fraisPort(double poids, boolean express, String pays) {
  double frais = 5;
  if (poids > 1) frais += 2;
  if (poids > 5) frais = 15;
  if (express) frais *= 1.5;
  if (pays.equals("FR")) frais -= 1;
  return frais;
}
```

Combien coûte un colis de **6 kg en express vers la France** ? **Personne ne sait** sans le lancer.

🛡️ Exemple : pinner le comportement avec des tests

On **lance** la méthode avec des entrées variées, on **note** ce qu'elle retourne, on **colle** les valeurs dans des `assertEquals`. On n'invente rien : on copie ce que la prod fait **déjà**.

🛡️ 5 tests qui figent les sorties observées

```
@Test void colis_leger_FR()       { assertEquals(4.0,  fraisPort(0.5, false, "FR")); }
@Test void colis_2kg_FR()         { assertEquals(6.0,  fraisPort(2.0, false, "FR")); }
@Test void colis_6kg_FR()         { assertEquals(14.0, fraisPort(6.0, false, "FR")); }
@Test void colis_6kg_express_FR() { assertEquals(21.5, fraisPort(6.0, true,  "FR")); }
@Test void colis_2kg_express_DE() { assertEquals(10.5, fraisPort(2.0, true,  "DE")); }
```

Sans ces tests, refactorer ce genre de cascade = **roulette russe** avec la production. Avec ces tests, on s'assure que le comportement reste identique.

🎯 Le pattern du TP4

Chaque exercice du TP4 suit le même squelette, avec **deux familles de tests** complémentaires : un filet anti-régression et un filet qui certifie le geste.

1. 📖 Lire

Survoler le code *smelly* sans chercher à **tout** comprendre. Repérer les zones qui semblent **bizarres** (longues, dupliquées, opaques).

2. ✅ Vérifier

Lancer les **caractérisations**. Toutes vertes ? On a un point de départ stable. Une qui échoue ? On corrige **avant** de toucher au reste.

3. 🔍 Identifier

Choisir **un seul** smell à traiter (Long Method, Magic Number...) et le refactoring associé du catalogue Fowler.

4. 🔧 Refactorer

Appliquer le refactoring via l'IDE (`Ctrl+.` / `F2`). Lancer les tests **à chaque pas**. Caract verte ? On continue. Rouge ? On annule.

5. 🔓 Débloquer

Retirer les `@Disabled` des tests de structure que le refactoring vient de **rendre vrais**. Ils passent : le geste est **fait**.

6. 💾 Commit

Commit dès que **tous** les tests passent. Petits commits = retour arrière facile. Message clair : `refactor: extract calculerTotalHT`.

⚠️ Si une caractérisation casse : **ne la modifiez pas**. Annulez la dernière transformation.

🌹 Le kata Gilded Rose

Créé par **Terry Hughes**, popularisé par **Emily Bache**. Un classique : **35 lignes** de code spaghetti à comprendre et faire évoluer **sans rien casser**.

🛍️ Le magasin

Chaque jour, les articles évoluent selon leur type :

- **Articles normaux** : perdent en qualité
- **Aged Brie** : gagne en qualité avec le temps
- **Sulfuras** : immuable, légendaire
- **Backstage passes** : règle complexe
- **Conjured** 🆕 : à **ajouter**

🎯 Votre mission

1. **Comprendre** le code via des *characterization tests*
2. **Refactorer** sans casser une seule caractérisation
3. **Ajouter Conjured** sur le code propre

L'`if` imbriqué qui faisait peur est devenu **trivial** à étendre.

La **situation typique** en entreprise : du code qu'on n'a pas écrit, sans docs, à faire évoluer. **Refactoring d'abord**, feature ensuite. 🎬 Démo live au TP4.

🌅 Où s'arrête le refactoring ?

Refactorer a un **coût** : du temps, un risque résiduel, une PR à relire. C'est une **décision**, pas un automatisme. Comment savoir si ça vaut le coup ?

✅ Ça vaut le coup quand...

- vous **allez toucher** cette zone dans les prochains mois
- le smell **ralentit** concrètement une feature ou un bug fix
- les **tests** prouvent que rien ne casse
- c'est **l'endroit précis** où vous intervenez

⚠️ C'est un piège quand...

- le code **marche**, est **stable**, personne ne le touche
- vous le faites juste parce que *"c'est pas joli"*
- il n'y a **pas de tests** : vous allez casser
- vous refactorez **et** ajoutez une feature en même temps

> 🛠️ L'artisan qui retourne son atelier **les jours** où il ne travaille plus, il range. On justifie un refactoring par un **bénéfice concret à venir**, pas par un idéal d'esthétique.

Quelques gestes à garder

Trois habitudes simples pour la suite : ranger un peu, nommer mieux, commenter moins.

🏕️ La règle du scout

Une règle adaptée du **scoutisme** au code par Robert C. Martin. Pas un grand refactoring : juste **un peu mieux** à chaque fois.

> 💬 *"Leave the campground cleaner than you found it."*
>
> **Laisse le campement plus propre que tu ne l'as trouvé.**
>
> - **Robert C. Martin**, *Clean Code* (2008)

🧹 Le geste quotidien

Chaque fois que vous **touchez** un fichier, laissez-le **un peu mieux** qu'à l'arrivée : un nom plus clair, une méthode extraite, un test ajouté.

📈 L'effet cumulé

20 développeurs qui améliorent **2 lignes** à chaque PR = la base de code **s'auto-nettoie**. C'est l'inverse de la spirale de la dette technique.

🛠️ Le geste de l'artisan qui **range son atelier** avant de partir. Pas pour la photo : pour **retrouver demain** un endroit où il peut travailler vite.

🏷️ Le nommage : première victoire

Avant de refactorer, **renommez**. Un bon nom **révèle l'intention** sans qu'on lise le corps. C'est le geste qui rend tout le reste plus facile.

> 💬 *"There are only **two hard things** in Computer Science: cache invalidation and **naming things**."* - **Phil Karlton**

🤦 Opaque

```
int d;
List<Integer> list;
void doStuff() { ... }
boolean flag = true;
```

✅ Parlant

```
int joursRestants;
List<Integer> agesEtudiants;
void envoyerRappel() { ... }
boolean estInscrit = true;
```

📦 Classe : nom commun

`Commande`, `Facture`

⚡ Méthode : verbe

`calculer`, `envoyer`

❓ Booléen : question

`estValide`, `aPaye`

💬 Les commentaires : utiliser avec parcimonie

Un commentaire qui dit **ce que fait le code** est un échec : il faut **renommer**. Un commentaire utile dit le **pourquoi**, le contexte que le code ne peut pas exprimer.

🙅 Commente pour compenser

```
// Vérifie si l'utilisateur a plus de 18 ans
if (u.getA() > 18) { ... }
```

Renommez plutôt : `if (utilisateur.estMajeur())`.

✅ Commente le pourquoi

```
// Le SDK distant exige un id positif strict ;
// 0 provoque un NPE côté serveur.
if (id <= 0) throw ...;
```

Information qui **ne peut pas** être déduite du code.

Avant d'écrire un commentaire : un **meilleur nom** ou une **méthode extraite** ne le rendraient-ils pas inutile ?

Pour la suite

Le prochain cap est simple : mettre ces gestes en pratique, d'abord à deux, puis sur du legacy.

📅 Ce qui vous attend

Trois étapes pour transformer ces concepts en **réflexes** : pratiquer en binôme, sécuriser du legacy, démontrer la démarche TDD seul·e sur feuille.

🥋 TP3 - Kata

5 kata en **pair programming** : années bissextiles, tennis, employés, pagination, yahtzee.

Driver / navigator, ping-pong TDD, rotation toutes les 5 à 10 min.

🧹 TP4 - Refactoring

6 exercices : Facture (Extract Method), CalculPrix (Magic Number), Menu (Extract Class), Animal (Polymorphisme), ServiceNotification (Parameter Object), **Gilded Rose**.

Caractérisation verte + tests `@Disabled` à débloquer.

💻 CC3 - Mini-kata

Mini-kata TDD **sur feuille**, 2 heures.

Pas d'IDE, pas de compilateur. Juste la **démarche TDD** appliquée à un problème simple.

À chaque étape, le même fil rouge : **petit pas, test d'abord, refactor ensuite**.

Des questions ?

Sébastien Nedjar

IUT d'Aix-Marseille - Département Informatique

🌐 github.com/IUTInfoAix-R203

✉️ sebastien.nedjar@univ-amu.fr

🎯 **Dès maintenant** : dans votre Codespace, essayez `F2` pour renommer un identifiant, puis sélectionnez quelques lignes et lancez `Ctrl+.` → *Extract to method*. Plus tôt vos doigts connaissent ces deux gestes, mieux vous vivrez les TP3 et TP4.

🛠️ **L'artisan, au fond, c'est vous.** Le CM1 a posé le *pourquoi* : du code dont on peut être fier, qu'on n'a pas peur de relire dans six mois. Ce CM2 a posé les *gestes* : tester avant, refactorer souvent, nommer juste, ranger en partant.