Artisanat logiciel, qualité et Git avancé

R2.03 - Qualité de développement

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

Git

TDD

Kata

Refactoring

Le module R2.03 en un coup d'œil

> Apprendre à produire du code **propre**, **testé**, **relu** et **maintenable**. Passer de "ça marche sur ma machine" à un vrai réflexe professionnel.

🔀

Git professionnel

Rebase, cherry-pick, PR, review

✅

TDD baby-steps

RED → GREEN → REFACTOR

🥋

Kata & pair programming

Driver, navigator, ping-pong

🧹

Refactoring

Code smells et transformations de Fowler

Organisation du module

CM1

Artisanat + Git + TDD intro

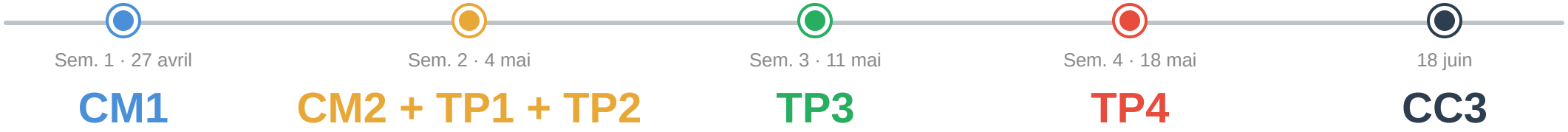
→ **TP1 Git + TP2 TDD**

CM2

TDD avancé + refactoring

→ **TP3 Kata + TP4 Refactoring**

Chaque CM prépare les **gestes concrets** mis en pratique dans les TP qui suivent.



Évaluation

Trois notes, un objectif : vérifier que vous **maîtrisez des gestes de métier**, pas juste que votre code tourne.

TP1 Git : mise à niveau, non noté. Le plus gros coefficient porte sur le **raisonnement TDD**, pas sur la vitesse de frappe.

Environnement de travail

Tout le module se fait sur **GitHub Codespaces** dans votre navigateur sans installation :

☕ **Java 25**

🧪 **JUnit 6 + AssertJ**

📦 **Maven (via mvnw)**

🔀 **Git + gh CLI**

🤖 **Copilot Chat**

📊 **ApprovalTests**

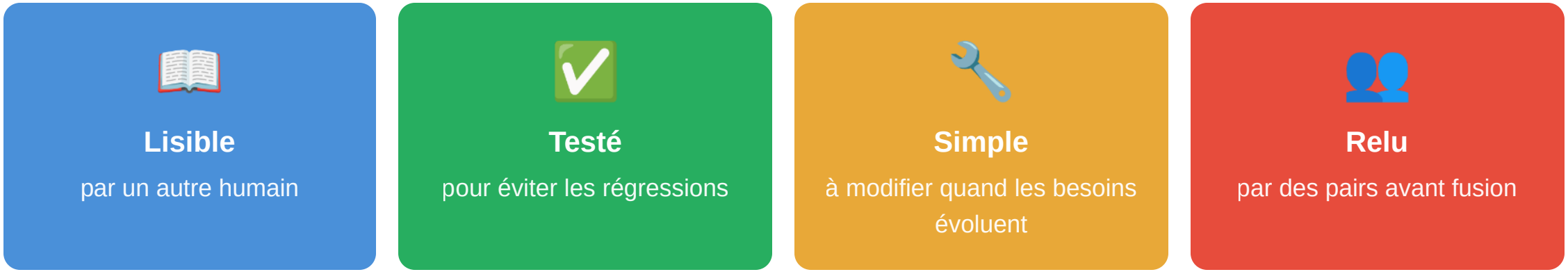
🌐 Dépôt étudiant : github.com/IUTInfoAix-R203/tp

Partie 1 - Artisanat logiciel

Pourquoi on parle de "qualité" ?

🔨 Qu'est-ce que l'artisanat logiciel ?

Un **artisan menuisier** livre une table **bien finie**, robuste, belle, facile à réparer. Un **artisan logiciel** livre du code qui doit être :



Le **"Software Craftsmanship"** est un mouvement né dans les années 2000, en réaction à l'industrialisation du monde du logiciel qui pressait les développeurs à livrer vite au détriment de la qualité.

🔨 Le manifeste de l'artisanat logiciel (2009)

✊ Snowbird, Utah, décembre 2008, en écho au Manifeste Agile de 2001, 4 principes, toujours sur le même modèle : **non seulement X, mais aussi Y**.

Non seulement du logiciel qui marche,

⚒️ **mais aussi du logiciel bien conçu.**

Non seulement répondre au changement,

📈 **mais aussi ajouter de la valeur régulièrement.**

Non seulement des individus et des interactions,

🤝 **mais aussi une communauté de professionnels.**

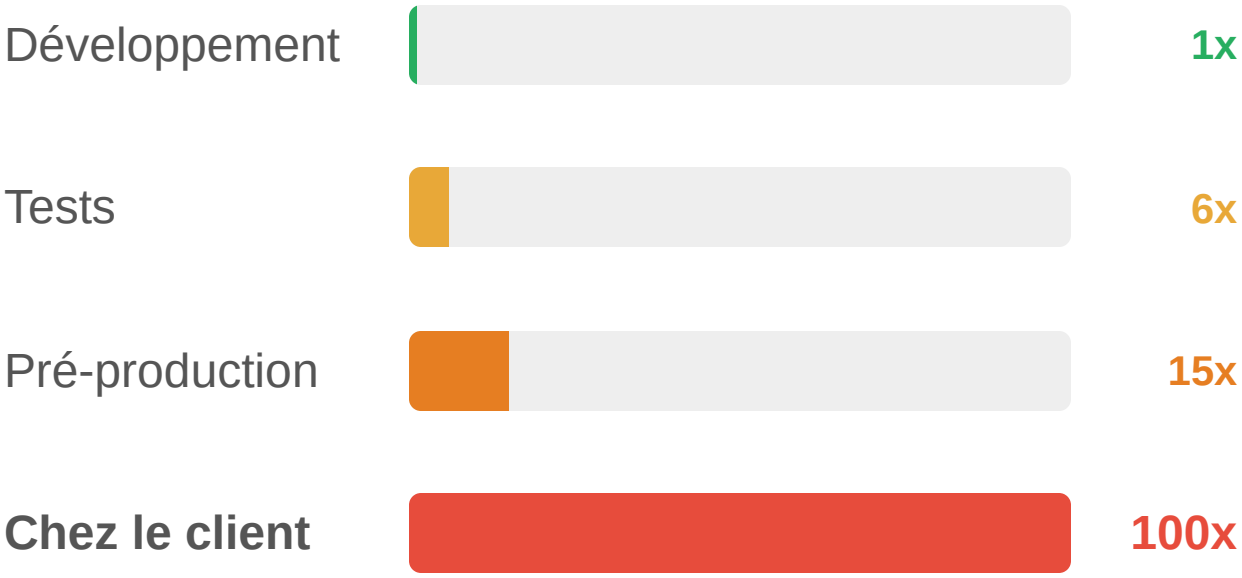
Non seulement collaborer avec le client,

🏛️ **mais aussi des partenariats productifs.**

💡 **L'idée à retenir** : le manifeste ne **rejette pas** l'agilité, il la **complète**. Les processus agiles ne suffisent pas : il faut aussi une exigence sur la **qualité du code** lui-même.

📉 Pourquoi c'est important : le coût des bugs

Plus un bug est détecté **tard**, plus il coûte cher à corriger. Ordre de grandeur (Boehm, 1981) :



💥 Knight Capital (2012)

Un bug de déploiement sur un système de trading a fait perdre **440 M$ en 45 minutes**.

Conséquence : **quasi-faillite et rachat en urgence**.

Cause : un drapeau mal réinitialisé dans un module obsolète.

➡️ La qualité du code n'est pas un luxe d'esthète : c'est une question de **survie économique**.

💳 La dette technique

Métaphore de **Ward Cunningham** (1992) : écrire du code sale pour livrer vite, c'est comme **emprunter** de l'argent. Ça aide à court terme, mais il faut **rembourser**, **avec intérêts**.

⚡ **Livrer vite**
gain court terme

→

💳 **Dette accumulée**
coût caché

→

🐌 **Ralentissement**
perte long terme

Quand la dette augmente :

⏳ Les features sont plus lentes à livrer

🐛 Les bugs se multiplient en production

😩 L'équipe se sent démotivée

💀 Le projet finit par mourir

🛠️ Les 4 gestes de l'artisan du logiciel

Derrière chaque pilier du module, un **geste métier** que pratique l'artisan, pas juste un outil technique.

🔀 Laisser une trace

Comme le menuisier qui date et signe ses ouvrages : on doit pouvoir revenir, comprendre, reprendre.

Git avancé, branches, PR, review → TP1

✅ Essayer avant de livrer

Comme la table qu'on essaie sur l'établi avant de la remettre au client : l'outil doit fonctionner avant de quitter l'atelier.

TDD, RED-GREEN-REFACTOR → TP2

🥋 Répéter pour apprendre

Comme l'apprenti qui refait cent fois le même geste : la main retient ce que la tête oublie.

Kata, pair programming, ping-pong → TP3

🧹 Entretenir ses outils

Comme le chef qui aiguise ses couteaux en fin de service : demain on s'en resservira, autant qu'ils coupent.

Refactoring, code smells, Fowler → TP4

🧭 Pourquoi ce module compte tout de suite

SAÉ 2.01 - Développement d'une application : Vous allez coder **en équipe**, **sur plusieurs semaines**, un projet réel. Les gestes de R2.03 - historique propre, tests, refactoring, review - y deviennent immédiatement utiles.

🔀

Git

pour collaborer sans se marcher dessus

✅

Tests

pour modifier sans paniquer

🧹

Refactoring

pour garder le projet vivable

🎯 Ce qu'on voit ici n'est pas "pour plus tard" : c'est ce qui rend un projet d'équipe tenable.

🎯 Compétences BUT visées

Trois apprentissages critiques du **PN BUT Informatique 2022** (annexe 15) sont couverts ici.

📐

AC11.02

Élaborer des conceptions simples

Compétence 1

Développer des applications informatiques simples

🧪

AC11.03

Faire des essais et évaluer leurs résultats

Compétence 1

Développer des applications informatiques simples

🔀

AC15.02

Mettre en place les outils de gestion de projet

Compétence 5

Identifier les besoins métiers et techniques des clients

Mots-clés officiels du PN : **Qualité**, **Test**, **Gestion de version**. C'est exactement le coeur de ce module.

Partie 2 - Git professionnel

Retour aux bases, puis les concepts avancés qui changent la vie.

🔀 Ce que vous savez déjà (plus ou moins bien)

Au S1, vous avez utilisé Git. Vérifiez avec moi :

✅ Je sais

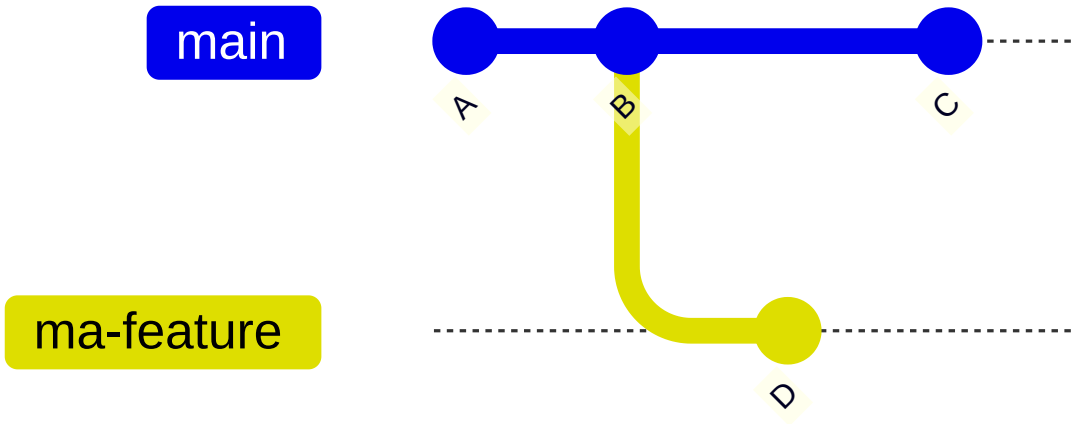
- `git clone`, `add`, `commit`, `push`
- Créer une branche
- Ouvrir un fork sur GitHub
- 😭 Pleurer quand un collègue écrase mes modifications
- 😭😭 Pleurer encore plus quand j'ai un conflit de fusion

🤔 Je ne sais pas (encore)

- Écrire un message de commit lisible 6 mois plus tard
- Intégrer une branche proprement (rebase vs merge)
- Rattraper un commit perdu sans paniquer
- Relire sérieusement la PR d'un pair

La moitié de ce TP1 corrige les mauvaises habitudes du S1. L'autre moitié, ce qui fait la différence en équipe.

🔀 Rappel express : le modèle Git



Il vous suffit de retenir **3 concepts** pour suivre le reste du cours :

📸

Commit

une photo du projet à un instant donné

🌱

Branche

une étiquette mobile qui avance avec vos commits

📍

HEAD

l'endroit où vous êtes en train de travailler

📝 Pourquoi un bon message de commit ?

Le message que vous écrivez **aujourd'hui** sera relu par **quelqu'un d'autre** (ou vous-même) dans 6 mois, pour comprendre pourquoi telle ligne a été écrite.

🤦 Mauvais messages

```
wip
fix
trucs
ca marche enfin putain
update file
```

✅ Bons messages

```
feat(auth): ajoute login via OAuth Google
fix(cart): corrige le total quand la remise est 0
docs(readme): detaille l'installation locale
refactor(facture): extrait appliquerTVA
```

🎯 Un message clair aujourd'hui = l'origine d'un bug se retrouve en 30 secondes dans 6 mois.

📝 Le format Conventional Commits

Un standard universel pour structurer vos messages et rendre votre historique lisible par tous.

```
<type>(<scope>): <description>
```

✨

`feat`

nouvelle fonctionnalité

🐛

`fix`

correction de bug

📝

`docs`

documentation

♻️

`refactor`

réorganisation sans changer le comportement

✅

`test`

ajout ou fix de tests

🔧

`chore`

maintenance, config, outils

🔄 GitHub Flow : le workflow en 4 étapes

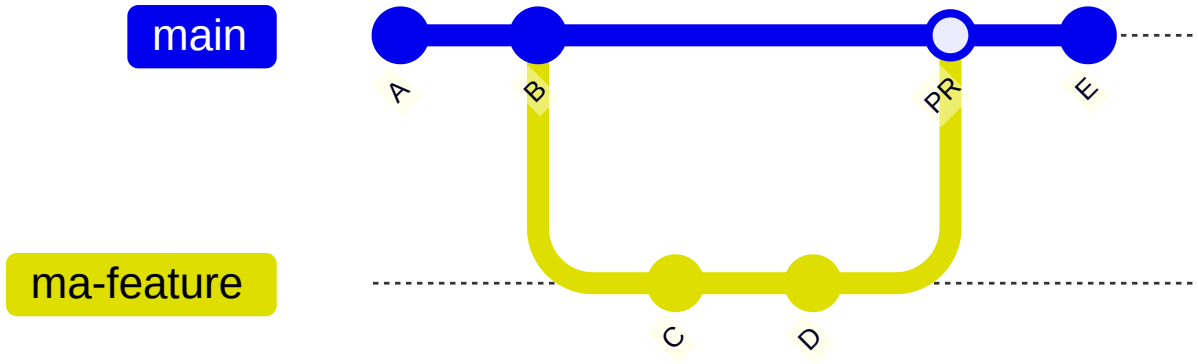
Le workflow standard en équipe, proposé par **Scott Chacon** (GitHub, 2011). Simple : une seule branche de référence (`main`) et des branches éphémères par fonctionnalité.

🌱 **1. Branche**
une par fonctionnalité

📸 **2. Commits**
atomiques + Conventional

🔍 **3. Pull Request**
dès qu'on est prêt à montrer

✅ **4. Merge**
après review



🎯 Règle d'or : **jamais de commit direct sur `main`**. Tout passe par une PR.

🔍 Pull Request : une conversation

Une PR n'est pas un formulaire administratif. C'est une **invitation à discuter** du code que vous proposez d'intégrer à la branche `main`.

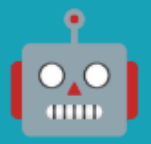
Ce qu'elle permet :

Comprendre

votre intention

Questionner

vos choix de conception



Vérifier

tests, qualité, conventions

Documenter

l'historique des décisions

Une PR qui vaut le coup : **un diff ciblé qui se lit, un contexte qui se comprend**.

🔍 Comment faire une bonne review ?

Reviewer, ce n'est pas dire "oui". C'est **lire le code avec les yeux d'un collègue dans 6 mois**, et mettre son nom en face de ce qui partira en prod.

✅ Checklist d'une bonne review

- Le code est-il **lisible** ?
- Les **noms de variable** sont-ils parlants ?
- Y a-t-il des **tests** ? Passent-ils ?
- Pas de `TODO` / `FIXME` orphelins ?
- La PR ne fait-elle qu'*une seule* chose ?

🤦 Ce qui n'est PAS une review

- **"LGTM"** sans lire le diff
- Approuver **sans tester** la branche en local
- Commenter les **fautes de frappe**, jamais le fond
- Valider sans attendre **la validation du CI**
- Bloquer la PR sur un **détail subjectif** (style personnel)

🎯 Une bonne review prend **15 minutes**. Un bug en prod prend **15 heures**.

🤖 Review automatique par Copilot

Dans vos Codespaces, **Copilot Code Review** peut commenter automatiquement une PR ouverte. Utile, mais ne remplace pas une vraie revue.

👁️ Ce qu'il fait bien

- Détecter les fautes de frappe
- Signaler les noms confus
- Proposer des simplifications
- Trouver des bugs

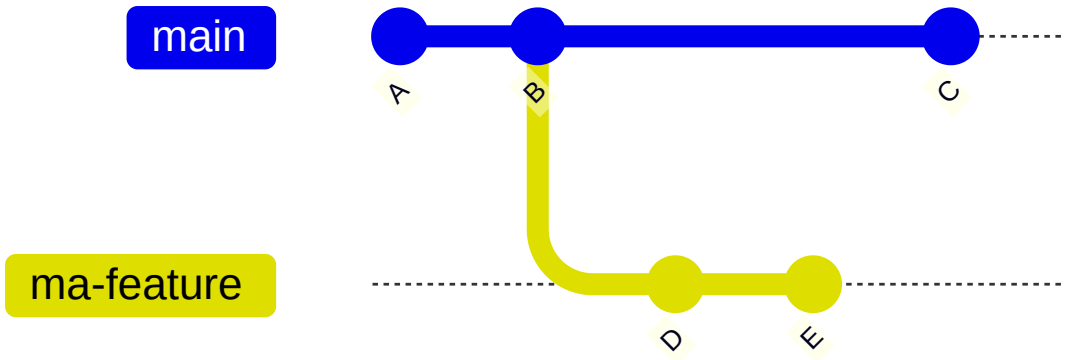
⚠️ Ses limites

- Ne comprend pas toujours l'intention métier
- Peut suggérer des choses hors sujet
- Peut **halluciner** un code faux qui semble bon
- Ne peut pas trouver tous les bugs

💡 **Copilot suggère** mais c'est **un humain qui décide.** Une vraie review commence par comprendre l'intention du contributeur et vérifier qu'elle est correctement mise en œuvre dans le code.

🔀 `git merge` : avant la fusion

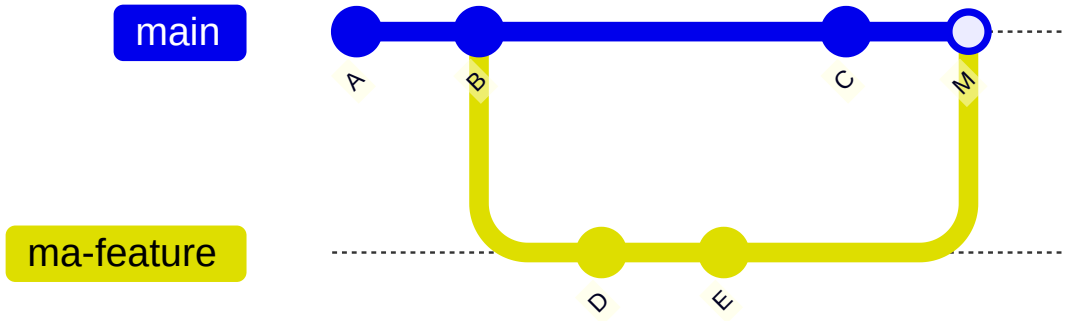
Vos commits sur la branche `ma-feature` sont prêts à être partagés. Voici la première façon de les intégrer à la branche `main` : `git merge`.



🕐 **Situation initiale** : la branche `main` continue d'avancer (commit C) pendant que la branche `ma-feature` travaille sur D puis sur E.

`git merge` : après la fusion

Vos commits sur la branche `ma-feature` sont prêts à être partagés. Voici la première façon de les intégrer à la branche `main` : `git merge`. Après cette opération, un nouveau commit de merge fait la jointure entre les deux branches.



✅ Avantages

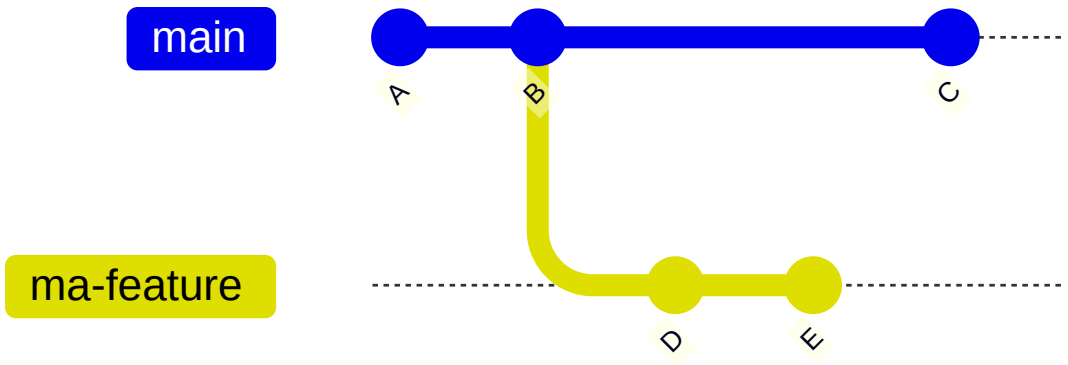
- Préserve l'**historique réel** du travail
- Les SHA des commits restent **stables** (partageables sans risque)

⚠️ Inconvénients

- Crée un **commit de merge** en plus
- Historique "**en arête de poisson**", plus difficile à relire

🔀 `git rebase` : avant le rebase

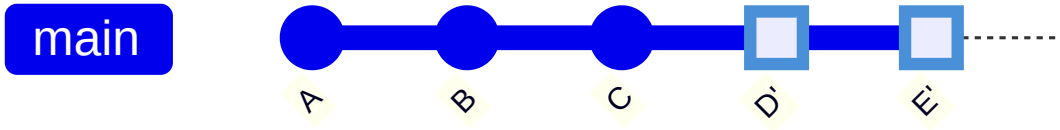
Vos commits sur la branche `ma-feature` sont prêts à être partagés. Voici la seconde façon de les intégrer à la branche `main` : `git rebase`.



🕐 **Même situation initiale** : on part de `main` = A-B-C et `ma-feature` = D-E.

🔀 `git rebase` : après le rebase

Vos commits sur la branche `ma-feature` sont prêts à être partagés. Voici la seconde façon de les intégrer à la branche `main` : `git rebase`. Le rebase **rejoue** les commits D et E sur la pointe de `main`. D et E deviennent D' et E' : même contenu, mais nouveaux commits avec de nouveaux SHA.



✅ Avantages

- Historique totalement **linéaire**, lisible comme une histoire
- Pas de commit de merge parasite

⚠️ Inconvénients

- **Réécrit les commits** : nouveaux SHA
- Dangereux sur une branche **déjà partagée** avec d'autres

⚖️ La règle d'or pour choisir entre merge et rebase

Rebase tes commits **tant que la branche n'est pas partagée** .
Merge dès qu'elle est **partagée** avec d'autres personnes.

Pourquoi ? Le rebase **change les SHAs**. Si votre coéquipier a déjà basé du travail sur votre branche publique, votre rebase lui fait **perdre ses commits**.

✅ OK

Je rebase ma branche personnelle `feat-login` avant de créer une PR.

💀 Danger

Je rebase `main` (partagée par toute l'équipe).

✏️ Rebase interactif : nettoyer l'historique

`git rebase -i HEAD~3` ouvre un éditeur qui liste les 3 derniers commits, un par ligne, avec une action modifiable devant chacun.

📝 Éditeur ouvert par `rebase -i`

```
pick a1b2c3 wip
pick d4e5f6 maj
pick g7h8i9 fix typo
# Rebase ...
# p, pick   = use commit
# r, reword = edit message
# s, squash = meld into previous
# f, fixup  = like squash, no msg
# d, drop   = remove commit
```

🎯 3 actions utiles au quotidien

- **`reword`** : corriger un message de commit
- **`squash`** / **`fixup`** : fusionner les petits commits de travail
- **`drop`** : retirer un commit indésirable

reword + **squash** + **drop** : 95% des cas d'usage.

✏️ Rebase interactif : exemple concret

Cinq commits de travail accumulés au fil de la journée, à fusionner en un seul commit propre prêt pour la PR.

🤦 Avant

```
a1b2c3 wip
d4e5f6 maj
g7h8i9 fix typo
h1j2k3 oh j'avais oublié un test
m4n5o6 deuxième essai
```

L'historique est illisible. Que s'est-il vraiment passé ?

🎯 Après rebase interactif

```
x7y8z9 feat(auth): ajoute login OAuth Google
```

Le message est clair. On sait ce qui a été ajouté et **pourquoi**.

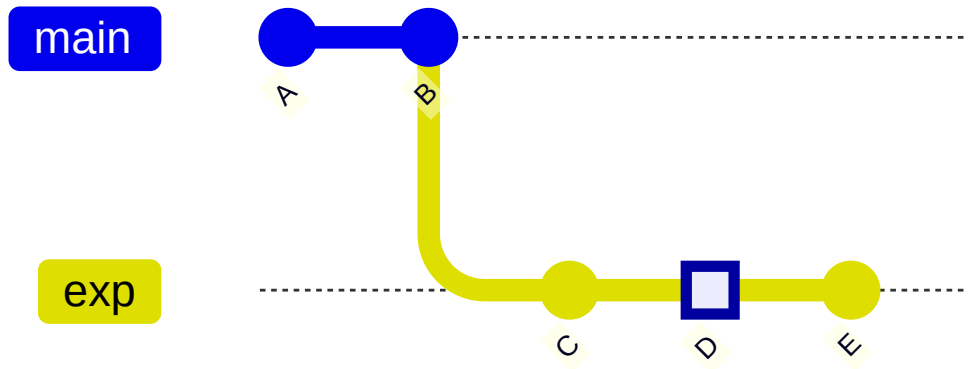
Refaire son historique avant la PR, c'est un acte de **respect** envers le relecteur.

🍒 `cherry-pick` : avant

Sur une branche expérimentale `exp`, le commit `D` contient un correctif utile. Vous voulez **juste ce commit-là** sur `main`, sans embarquer `C` ni `E`.

Le besoin :

- Récupérer **uniquement D** sur `main`
- Laisser `C` et `E` sur `exp` (encore en cours)
- Sans `merge` (qui rapatrierait tout)
- Sans `rebase` (qui réécrirait `exp`)



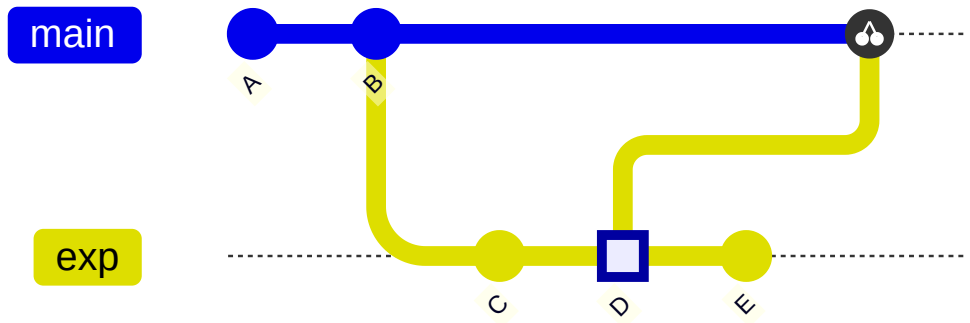
🕐 **Situation initiale** : `main` = A-B, `exp` = C-D-E.

🍒 `cherry-pick` : après

`cherry-pick` **copie** le commit `D` sur la pointe de `main`. Il devient `D'` : même contenu, mais nouveau commit avec un **nouveau SHA**.

```
git checkout main
git cherry-pick D
```

`exp` reste **intacte** : C, D et E sont toujours là pour continuer la branche expérimentale plus tard.



⚠️ Cherry-pick **copie** le commit : vous récupérez l'idée, pas le même SHA.

🆘 Outil de secours : reflog

Vous tapez `git reset --hard HEAD~5` et réalisez que vous venez d'effacer 5 commits. **Pas de panique** : Git garde une mémoire locale de tous vos déplacements.

1️⃣ Lister les déplacements

```
$ git reflog
a1b2c3 HEAD@{0}: reset: moving to HEAD~5
g7h8i9 HEAD@{1}: commit: feat: ajoute X  ←
d4e5f6 HEAD@{2}: commit: wip
...
```

2️⃣ Revenir au commit perdu

```
$ git reset --hard g7h8i9
HEAD is now at g7h8i9 feat: ajoute X
✅ Vos commits sont récupérés.
```

`reflog` est votre **filet de sécurité** : tant qu'un commit a existé localement, il reste récupérable pendant ~30 jours.

⚠️ Commandes destructrices et leurs alternatives

Certaines commandes Git détruisent du travail **sans demander confirmation**. Pour chacune, il existe une variante équivalente qui refuse d'écraser silencieusement.

💀 Dangereux	🛡️ Plus sûr
`git reset --hard`	`git reset`
Perd toutes les modifications non commitées.	Déstage sans toucher au contenu (mode `--mixed` par défaut).
`git push --force`	`git push --force-with-lease`
Écrase la branche distante, même si un collègue a poussé entre-temps.	Refuse le push si quelqu'un a poussé entre-temps pour ne pas perdre son travail.

Règle de survie : **privilégiez toujours la variante qui refuse d'écraser silencieusement**.

🎬 Démo live : rebase d'une branche sur main

Cycle complet en direct : **brancher, commiter, rebaser, pousser sans casser le travail des autres**. Posez vos questions pendant la démo.

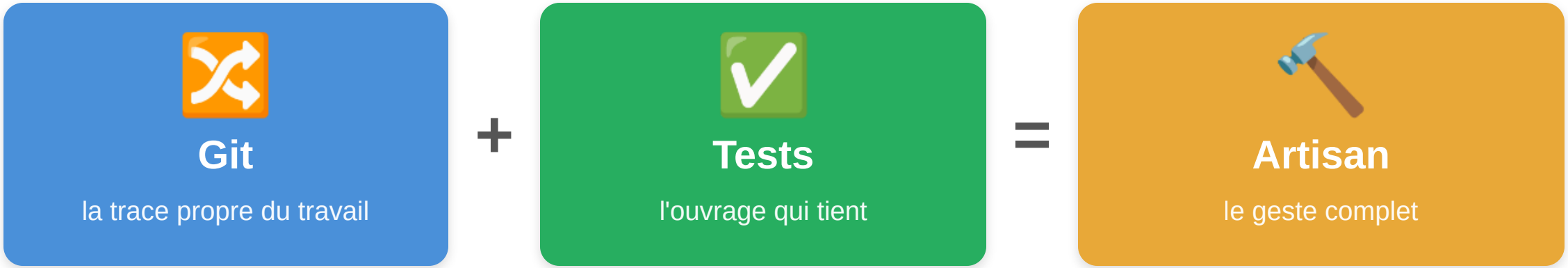
1. Clone `IUTInfoAix-R203/tp1` dans un Codespace
2. Création de la branche `feat-exemple` + deux commits
3. Deux commits sur `main` pour simuler le travail d'un collègue
4. Rebase sur `main` : `git rebase origin/main`
5. Visualisation : `git log --oneline --graph --all`
6. Push sécurisé : `git push --force-with-lease`

⚠️ **Mise en scène pédagogique** : je vais volontairement simuler un commit mal formé pour montrer comment le rebase interactif le corrige. **Ne reproduisez pas ce style en vrai.**

👁️ Observez : l'historique de `main` reste **linéaire** et **lisible** avant et après le push.

🔗 De l'historique propre au code qui tient

Git assure la **trace** propre de votre travail. Mais une belle histoire ne garantit pas que **le code actuel marche**. Il vous faut l'autre moitié du métier.



Vérifier son ouvrage **à chaque geste**, pas à la livraison. C'est là qu'intervient le **TDD**.

Partie 3 - Introduction au TDD

Quand tester un logiciel vous aidera à mieux le concevoir !

🐛 Le problème du code "ça marche sur ma machine"

Trois situations qu'on a toutes et tous vécues, et qui ont **une seule cause profonde**.

📦 Vous livrez un projet qui marchait chez vous, et il **plante** chez le prof.

🔄 Vous corrigez un bug et un **autre** que vous aviez résolu il y a 3 jours **réapparaît**.

😰 Vous modifiez une fonction en ayant **peur** que ça casse ailleurs.

La cause profonde : **aucun filet de sécurité** qui vous prévient quand vous cassez quelque chose, vous n'avez aucune assurance que votre logiciel continue à fonctionner comme attendu.

🤖 Qu'est-ce qu'un test automatisé ?

Un test automatisé, c'est **du code qui exécute d'autres bouts de code** et vérifie automatiquement le résultat attendu. Pas de clic manuel, pas de vérif à l'œil.

👀 Test manuel

- Lancer l'appli, cliquer, regarder l'écran
- **Lent** : refaire chaque vérification à la main
- **Oubliable** : on ne re-teste que ce qu'on pense à re-tester
- **Pas reproductible** entre deux personnes ou deux moments différents

🤖 Test automatisé

- Du code qui appelle votre code et compare le résultat
- **Rapide** : des centaines de vérifications en quelques secondes
- **Répétable** : relançable à volonté, à l'identique
- **Déterministe** : même résultat à chaque exécution

Le test automatisé c'est un robot assistant invisible qui **refait toutes vos vérifications en quelques secondes**, à chaque fois que vous lui demandez.

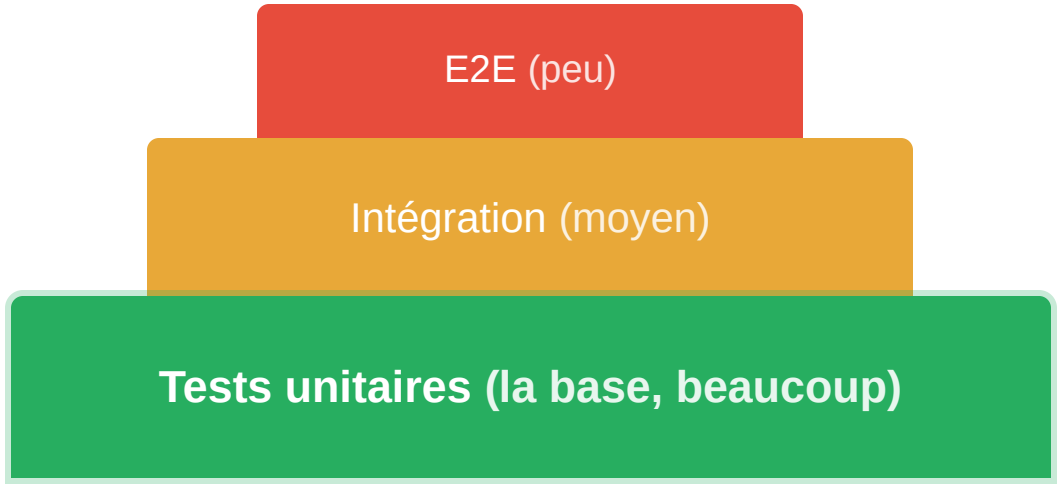
🎯 Le test unitaire : zoomer sur une seule unité

Un test **unitaire** vérifie **une seule unité de code** (une méthode, ou une classe), **isolée** du reste de l'application : pas de base de données, pas de réseau, pas de fichiers.

⚡ **Rapide** : moins d'une seconde par test

🧪 **Isolé** : aucune dépendance externe

🔢 **Nombreux** : souvent des centaines par projet



👉 La pyramide des tests, détaillée au **CM2**

En **TP2**, **TP3** et **TP4**, vous verrez **uniquement** des tests unitaires.

📝 À quoi ressemble un test JUnit ?

Un test JUnit, c'est **juste une méthode Java** qui en appelle une autre et vérifie le résultat. Trois phases, toujours les mêmes : **Given · When · Then**.

📄 **CalculatriceTest.java**

```java
@Test
void additionne_deux_et_trois_retourne_cinq() {
    // Given : on a une calculatrice
    Calculatrice calc = new Calculatrice();
    // When : on additionne 2 et 3
    int resultat = calc.additionne(2, 3);
    // Then : on attend 5
    assertThat(resultat).isEqualTo(5);
}
```

🛠️ **Given** : préparer les objets dont on a besoin

▶️ **When** : appeler la méthode qu'on veut tester

✅ **Then** : vérifier que le résultat correspond à l'attendu

📛 Le **nom du test** raconte ce qu'on vérifie

Pas de magie : du **pur Java**, une annotation `@Test`, et un `assertThat(...)` qui décide.

▶️ Lancer ses tests : le feedback loop

Vous lancez vos tests d'une commande ou d'un clic, et JUnit vous répond **en quelques secondes** : tout vert, ou rouge avec le détail précis de l'échec.

💻 En ligne de commande

```
$ ./mvnw test
```

🖱️ Dans l'IDE (VS Code, Codespace)

Clic sur l'icône ▶ dans la marge à côté d'`@Test`, ou clic droit sur la méthode → *Run Test*.

⏱️ Tout vert

```
[INFO] Tests run: 12, Failures: 0
[INFO] BUILD SUCCESS
```

🐛 Un test échoue

```
[ERROR] additionne_deux_et_trois...
  expected: 5
   but was: 6
[ERROR] BUILD FAILURE
```

🔁 Code → tests → résultat **en quelques secondes** : c'est le **feedback loop**. Plus il est court, plus vous l'exécutez souvent et plus vous codez en confiance.

🪢 La suite de tests : votre filet de sécurité

Un test isolé, c'est un seul point d'ancrage. Ce qui vous **tient vraiment**, c'est la **suite** qui s'accumule au fil du projet, et qui se relance **en entier** à chaque modification.

1 test isolé	10 tests accumulés	100+ tests (la suite)
🪝	🪝🪝🪝	🪝🪝🪝🪝🪝🪝🪝
Couvre **un seul cas**. Tout le reste du code peut casser sans alerte.	Un **filet qui prend forme**. Les régressions sur les cas testés sont attrapées.	Filet **dense**. Plus votre filet devient efficace, plus vous modifiez votre code **sereinement**.

C'est l'**accumulation** qui transforme des tests isolés en **filet de sécurité**. Grâce à votre suite de tests, vous pouvez coder en confiance.

🤖 La CI : le filet qui se déploie automatiquement

À chaque `git push`, GitHub Actions relance votre suite de tests **sans même que vous demandiez**. C'est l'**Intégration Continue (CI)**.

📤 **git push** sur GitHub

⬇

🤖 **GitHub Actions** s'active automatiquement

⬇

⏱️ **Toute la suite** de tests s'exécute

⬇

🚦 **Statut visible** sur la Pull Request

⏱️ **All checks have passed**

✓ **Maven build** : successful in 12s

🐛 **Some checks were not successful**

✗ **Maven build** : failed in 8s

🛡️ Le filet **se déploie tout seul**. Plus moyen de pousser du code cassé sans le savoir.

🌟 Qu'est-ce qu'une bonne suite de tests ?

Toutes les suites de tests ne se valent pas. Une **bonne** suite tient sur **quatre propriétés**, et chacune compte autant que les autres.

🎯 **Pertinente** : couvre les cas qui comptent : nominal, limites, erreurs.

📖 **Lisible** : un humain comprend ce qui est testé en lisant le test.

🚀 **Rapide** : on doit pouvoir la lancer souvent, sans attendre.

🧱 **Maintenable** : on peut faire évoluer le code sans tout réécrire.

Une suite mal calibrée devient un **fardeau** au lieu d'un filet : l'équilibre entre les quatre est délicat.

⚖️ Le piège de tester APRÈS avoir codé

Quand le code marche déjà, écrire des tests devient fastidieux. Et on ne sait pas **quand s'arrêter**, parce qu'aucun test n'est *nécessaire* par construction. Deux dérives symétriques s'installent.

😴 Suite anémique

"Le code marche, j'ajouterai les tests plus tard..."

- La suite **stagne**, le filet ne se forme jamais
- Régressions silencieuses à chaque modification
- La **peur** de modifier revient

🐘 Suite obèse

"Dans le doute, je teste tout, encore et encore..."

- Suite **lente**, on la lance moins souvent
- Tests **fragiles** qui cassent à la moindre refonte
- Devient un **frein** à l'évolution du code

💡 Et si chaque petit bout de code **naissait** d'un test qui en a besoin ? Le test ne se contenterait pas de précéder le code : il **dirigerait** son écriture, un pas à la fois.

✅ TDD : l'idée contre-intuitive

Le **D** de TDD, c'est **Driven**, *dirigé*. Un test à la fois **commande** ce que vous codez ensuite. La conception **émerge pas à pas**, elle n'est pas planifiée à l'avance.

🚀 Pourquoi c'est puissant

- Le test **précise** le comportement attendu, ligne par ligne
- Vous n'écrivez que le code **strictement nécessaire**
- Le filet de sécurité **se construit en même temps** que le code
- L'API est **guidée** par l'usage : pénible à tester = mal pensée

🤔 Pourquoi c'est contre-intuitif

Notre réflexe : *"j'écris le code, je testerai si j'ai le temps"*. On finit par tester **ce qu'on a codé**, au lieu de **ce qu'il fallait faire**.

TDD inverse la logique : on exprime **un besoin** sous forme de test, puis on code le minimum pour y répondre. Et on recommence.

⚠️ TDD ≠ **Test First** (écrire tous les tests d'avance, puis coder). TDD c'est **un test à la fois**, suivi d'un petit pas de code, suivi d'un nettoyage. Une procédure précise, qu'on va découvrir maintenant.

🔁 Le cycle RED - GREEN - REFACTOR

Trois étapes, dans cet ordre, pour chaque petit comportement à ajouter. À la fin du tour, on reprend avec le test suivant.

1. 🔴 RED

Écrire **un test qui échoue**. C'est la preuve qu'il vérifie vraiment quelque chose. Ne **pas coder** avant d'avoir vu le rouge.

2. 🟢 GREEN

Écrire le **minimum** de code pour que le test passe. Oui, même si c'est moche. **Fake it** si besoin.

3. 🧹 REFACTOR

Nettoyer le code **sans casser les tests**. C'est le moment d'extraire, renommer, simplifier.

🔁 ... puis on **reprend en 1.** avec le test suivant : un petit pas de plus dans la conception.

Chaque tour ajoute **un petit bout de comportement validé**. La conception **émerge** tour après tour.

🎣 Les 3 stratégies de Kent Beck

Pour passer du **RED** au **GREEN**, trois approches au choix selon la complexité du test à faire passer.

🎭 Fake it

Retourner une **valeur en dur** qui fait passer le test.

```
// saluer("Alice") == "Hello, Alice"
return "Hello, Alice!";
```

💡 **Premier réflexe**. Toujours commencer ici.

📐 Triangulation

Un **2e test** avec une autre valeur vous oblige à généraliser.

```
// saluer("Alice"), saluer("Bob")
return "Hello, " + nom + "!";
```

💡 Quand **fake it** ne suffit plus.

💡 Obvious

L'implémentation tient en **une ligne évidente**.

```
// additionner(2, 3) == 5
return a + b;
```

💡 **Rare**. OK si la solution est vraiment triviale.

Toujours commencer par **fake it**. Trianguler quand un seul cas ne suffit plus. Aller à l'évidence si elle est vraiment évidente.

👣 Baby steps : petits pas, toujours

La **règle d'or du TDD** : un test = un petit pas de code = un point de contrôle. Si vous modifiez 30 lignes entre deux exécutions de tests, vous codez **à l'ancienne**, en priant.

🎯 Bon rythme

- Activer **1 test** → le voir rouge (15 s)
- Écrire **3 lignes** → le voir vert (15 s)
- **Refactor** si besoin → suite verte (15 s)
- Tour suivant

💥 Mauvais rythme

- Activer **5 tests** d'un coup
- Écrire **40 lignes** sans tester
- Lancer la suite : 3 verts, 2 rouges
- Débugger **sans savoir** qui a cassé quoi

Plus le pas est petit, plus le **diagnostic** en cas d'échec est immédiat. C'est tout l'intérêt.

🎬 Kata live : HelloWorld en TDD

On va écrire **ensemble** une méthode `saluer(String nom)`. Quatre tests fournis, activés **un par un**, dans l'ordre. Vous observez, je code.

📋 Spécification

La méthode `saluer(String nom)` retourne :

- `"Hello, World!"` si `nom` est `null` ou vide
- `"Hello, <nom>!"` sinon

🧪 Les 4 tests à activer (dans l'ordre)

1. `saluerSansNomRetourneHelloWorld()`
2. `saluerChaineVideRetourneHelloWorld()`
3. `saluerAliceRetourneHelloAlice()`
4. `saluerBobRetourneHelloBob()`

👁️ Observez le rythme : **activer → rouge → écrire le minimum → vert → refactor → tour suivant.**
Pas de code en avance.

🤖 Copilot Chat : votre tuteur, pas votre code-monkey

Sur les TP de R2.03, Copilot est configuré comme **tuteur TDD**. Il aide à raisonner, mais **refuse** de court-circuiter la démarche.

🎓 Ce que Copilot fera

- Expliquer un concept TDD
- Vous orienter vers la Javadoc
- Vous aider à débloquer une impasse
- Refuser de coder si un test n'est pas encore activé

🙅 Ce qu'il refusera

- *« Écris tout le TP à ma place »*
- Donner une solution dès le premier prompt
- Coder avant que vous ayez vu le rouge
- Donner les commandes Git si l'exercice n'est pas terminé

Copilot sera votre **tuteur pendant les TP** mais n'oubliez pas qu'au CC3, vos réflexes devront être **les vôtres**.

🎯 Ce qu'on fait maintenant

Mettre en pratique les **deux moitiés** du cours d'aujourd'hui.

🔀 TP1 : Git avancé

2 h : non noté

Historique lisible, PR, review, intégration propre.

✅ TP2 : TDD

4 h : noté (CC1)

Cycle RED-GREEN-REFACTOR, fake-it, triangulation, ApprovalTests.

Dès le prochain TP : commits relisibles, PR comme une conversation, et un test qui vous dit quand vous avez cassé quelque chose.

👉 Classroom : **github.com/IUTInfoAix-R203/tp1**

À suivre : CM2

On approfondira le TDD, on découvrira le **pair programming**, et on préparera le **refactoring** du TP4.

TDD avancé

Kata & pair programming

Refactoring

📅 Sem. 2 · 4 mai 2026

Des questions ?

Sébastien Nedjar

IUT d'Aix-Marseille - Département Informatique

🌐 github.com/IUTInfoAix-R203/tp1

✉️ sebastien.nedjar@univ-amu.fr

🎯 **Dès maintenant** : dans votre Codespace, faites un petit commit, ouvrez une PR, et lancez `./mvnw test`. Plus tôt vos doigts connaissent ces gestes, plus tôt le métier rentre.

🛠️ **L'artisan, au fond, c'est vous.** Ce CM1 a posé le *pourquoi* - un historique qu'on relit sans honte, un code qui dit *quand* il casse. Le TP1 et le TP2 vont transformer ces idées en **gestes quotidiens**. On n'aiguise pas sa scie le jour où on doit couper la planche ; on l'aiguise la veille.