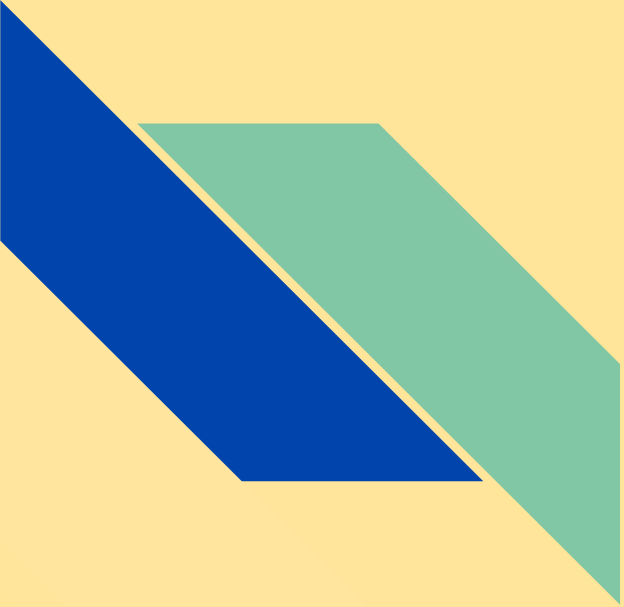

Introduction au réusinage de code

Plan de séance:

- ❏ Introduction au code propre
- ❏ Noms de variable
- ❏ Fonctions
- ❏ Commentaires

Introduction au code propre

Bjarne Stroustrup, C++

J'aime que mon code soit **élégant et efficace**. La logique doit être **simple** pour qu'il soit difficile pour les bogues de s'y cacher, les dépendances **minimales** pour faciliter la maintenance, la gestion des erreurs **complète** et conforme à une stratégie bien définie, et la performance **proche de l'optimale** pour ne pas inciter les gens à désordonner le code avec des optimisations non maîtrisées.

Le code propre ne fait qu'une seule chose, et il la fait bien.

“Big” Dave Thomas, Eclipse

Un **code propre** peut être lu et amélioré par un développeur autre que son auteur original. Il est doté de **tests unitaires et d'acceptance**. Il utilise des **noms significatifs**. Il ne propose qu'**une seule façon** de faire une chose plutôt que plusieurs. Il a des **dépendances minimales**, lesquelles sont définies explicitement, et il fournit une **API claire et minimale**. Le code doit être **documenté** (littéraire) car, selon le langage, toutes les informations nécessaires ne peuvent pas être exprimées clairement dans le code seul.

Ward Cunningham, Wiki

Vous savez que vous travaillez sur du **code propre** lorsque chaque routine que vous lisez s'avère être à peu près ce à quoi vous vous attendiez. Vous pouvez parler de **beau code** lorsque celui-ci donne également l'impression que le langage a été conçu spécifiquement pour résoudre ce problème.

Noms de variable

Noms Révélant l'intention

Règle Clé : Le nom d'une variable, fonction ou classe doit **répondre aux questions** : pourquoi elle existe, ce qu'elle fait et comment elle est utilisée. **Un bon nom ne nécessite pas de commentaire pour être compris.**

Principe d'Explicitation : Pour les mesures, incluez ce qui est mesuré et l'**unité de mesure** dans le nom pour expliciter le contexte. int joursDepuisModification est bien formé, nDaysBefore, beaucoup moins.

Évocateur: Il faut que chaque variable ait un contexte explicite dans le code.

Noms révélant l’intention

DIRTY CODE

```
def get_them():
 l = []
 for x in the_list:
  if x[0] == 4:
   l.append(x)
 return l
```

CLEAN CODE

```
def get_cellules_signalees():
 cellules_signalees = []
 for cellule in toutes_les_cellules:
  STATUT_SIGNALEMENT = 4
  if cellule[STATUT_INDEX] ==
STATUT_SIGNALEMENT:
   cellules_signalees.append(cellule)
 return cellules_signalees
```

Éviter la Désinformation

Éviter les Faux Indices : N'utilisez pas de mots ou d'acronymes dont la signification est déjà **établie** dans le contexte technique ou commercial, car cela obscurcit l'intention réelle du code.

Attention aux Similarités : Les noms de concepts similaires doivent être **écrits de manière cohérente**, mais les différences doivent être **évidentes** pour éviter la confusion lors de la lecture ou de l'utilisation de l'autocomplétion.

Typographie Ambigüe : Proscrire les noms de variables qui ressemblent trop à des constantes (ex: `l` minuscule ou `O` majuscule qui ressemblent à `1` et `0`).

Éviter la Désinformation

DIRTY CODE

```
def get_comptes_list()
 comptes_list = [c for c in DB if c.actif]
 return comptes_list
```

CLEAN CODE

```
def get_comptes_actifs()
 comptes_actifs = [c for c in DB if c.actif]
 return comptes_actifs
```

Faire des Distinctions Significatives

Les noms doivent être différents parce qu'ils **signifient quelque chose de différent**. Évitez les "mots de bruit" (Noise Words) ou les séries numériques qui différencient les noms sans apporter d'information nouvelle (ex: `Info`, `Data`, `a1`, `the_`).

Le développeur ne devrait jamais avoir à deviner la différence entre des noms similaires (ex: `get_account()` vs `get_account_info()`). Les distinctions doivent être **significatives** et **fonctionnelles**.

Faire des Distinctions Significatives

DIRTY CODE

```
class Client:
  pass

class ClientObject:
  pass

def get_compte():
  pass

def get_compte_info()
  pass
```

CLEAN CODE

```
class Client:
  pass

class ResumeClient:
  pass

def get_compte():
  pass

def get_details_paiement_compte:
  pass
```

Prononçable, rechercheable

Prononçabilité : Les noms doivent être **facilement prononçables** pour permettre une discussion fluide et intelligente en équipe sans recourir à des acronymes bizarres ou des sons étranges.

- *(Ex: `genymdhms` doit devenir `generationTimestamp`).*

Recherchabilité : Évitez les noms d'une seule lettre ou les **constantes numériques** (nombre magique) dans le code. Les noms longs et descriptifs sont faciles à rechercher dans un projet (ex: `grep`), contrairement à `7` ou à la variable `e`.

Mental mappping

Éviter la Traduction Mentale : N'utilisez pas de noms qui forcent le lecteur à les **traduire mentalement** vers le concept réel qu'ils représentent. Choisissez des noms tirés du **domaine du problème** (métier) ou du **domaine de la solution** (technologie).

Clarté sur l'Intelligence : Un **développeur professionnel** privilégie la **clarté** avant tout. Évitez les noms cryptiques ou les abréviations complexes qui démontrent votre intelligence mais nuisent à la compréhension collective.

Exceptions Traditionnelles : L'usage de variables à une seule lettre comme **compteurs de boucle** (`i`, `j`, `k`) est acceptable **uniquement** si leur portée est extrêmement réduite, car c'est une convention établie.

Mental mapping

DIRTY CODE

```
def extraire_ressource(url):
  r = url.split("://")[1]
 r = r.split("/", 1)[1]
 return r

def calculer_somme(a, b):
 c = a + b
 return c
```

CLEAN CODE

```
def extraire_ressource(url_complet):
 url_sans_protocole =
url_complet.split("://", 1)[1]
 chemin_ressource =
url_sans_protocole.split("/", 1)[1]
 return chemin_ressource

def calculer_somme(valeur_debit,
valeur_credit):
 total = valeur_debit + valeur_credit
 return total
```

Un seul mot par concept

Règle Clé : Choisissez **un seul mot** pour désigner un concept abstrait et tenez-vous-y dans l'ensemble de votre base de code. Cela s'applique aux noms de **méthodes** et aux noms de **classes/composants**.

Problème Majeur : Évitez d'utiliser des synonymes équivalents (ex: `fetch`, `retrieve`, `get`) pour la même action dans différentes classes. Cela force les développeurs à **mémoriser quel nom va avec quelle classe**.

Pour les Composants : Maintenez une distinction claire entre les rôles des composants (ex: `Controller` vs `Manager` vs `Driver`). Si deux composants font des choses similaires, utilisez une **terminologie cohérente** pour leur rôle.

Noms du domaine de solution

Parlez Technique (Domaine Solution) : Utilisez les **termes de l'informatique** (CS), les noms d'**algorithmes** ou de **patrons de conception** (`Visitor`, `JobQueue`). N'hésitez pas à employer le jargon technique, car votre public est composé de programmeurs.

Parlez Métier (Domaine Problème) : Si un concept n'a pas d'équivalent technique standard (pas de jargon de programmeur), utilisez les noms tirés du **domaine métier** ou du client.

Séparer les Concepts : Un bon code sépare les concepts du domaine **Solution** (comment le code fonctionne) et du domaine **Problème** (ce que le code fait pour le client), en utilisant les noms appropriés à chaque contexte.

Fonctions

Petites fonctions

Règle Clé : Les fonctions doivent être **courtes** et très bien définies. Idéalement, elles ne devraient pas dépasser les 20 lignes et, si possible, ne faire que quelques lignes.

Problème Majeur : Une fonction longue cache souvent plusieurs niveaux d'abstraction et plusieurs responsabilités, ce qui la rend difficile à lire, à tester et à maintenir.

Structure : La structure d'une fonction doit ressembler à un **récit** : les fonctions de haut niveau appellent celles de niveau inférieur.

Une fonction fait une chose

Règle Clé: Une fonction fait "une seule chose" si toutes ses étapes sont à **un seul niveau d'abstraction** en dessous du nom de la fonction. Elle décompose un concept de haut niveau en étapes immédiatement subordonnées.

Test Narratif (Le Paragraphe "TO") : Si vous pouvez décrire la fonction avec un court paragraphe "POUR..." (TO) où les étapes correspondent directement aux appels internes, elle est probablement "propre".

Test d'Extraction : Une fonction fait plus d'une chose si vous pouvez en extraire une sous-fonction dont le nom **n'est pas une simple reformulation** de l'implémentation extraite.

Symptôme : La division d'une fonction en sections (`déclarations`, `initialisations`, `logique`) est un **symptôme clair** qu'elle fait plus d'une chose.

Dirty code

```
# La fonction gère la récupération, la validation et le formatage.
def process_data_and_save(data_id):
    # 1. Niveau 1 (Récupération de bas niveau)
    raw_data = database.fetch_record(data_id)

    # 2. Niveau 2 (Validation et Logique métier)
    if raw_data is None or raw_data['status'] != 'active':
        raise ValueError("Data non trouvée ou inactive.")

    # 3. Niveau 3 (Transformation/Formatage)
    formatted_data = {
        'identifier': data_id,
        'name_upper': raw_data['name'].upper()
    }

    # 4. Niveau 4 (Envoi à un autre service)
    external_service.send(formatted_data)

    return formatted_data
```

Clean code

```
# Chaque fonction a une seule responsabilité claire.

def fetch_active_record(data_id):
    record = database.fetch_record(data_id)
    if record is None or record['status'] != 'active':
        raise ValueError("Record non trouvé ou inactif.")
    return record

def format_record_for_export(record, data_id):
    return {
        'identifier': data_id,
        'name_upper': record['name'].upper()
    }

def execute_data_pipeline(data_id):
    # La fonction orchestratrice (un seul niveau d'abstraction)
    active_record = fetch_active_record(data_id)
    formatted_data = format_record_for_export(active_record, data_id)
    external_service.send(formatted_data)
    return formatted_data
```

Niveaux d’abstraction

Règle Clé : Toutes les instructions à l'intérieur d'une fonction doivent se situer au **même niveau d'abstraction**. Ne mélangez jamais les concepts de haut niveau (essentiels) avec les détails de bas niveau (implémentation).

La Règle de l'Étape Descendante (The Stepdown Rule) : Le code doit se lire comme une **narration "du haut vers le bas"**. Chaque fonction doit être immédiatement suivie par les fonctions qu'elle appelle, lesquelles se situent au niveau d'abstraction immédiatement inférieur.

Symptôme de Confusion : Le mélange des niveaux d'abstraction (ex: un appel à `getHtml()` de haut niveau juxtaposé à un détail comme `.append("\n")` de bas niveau) rend le code confus et encourage l'ajout de détails non pertinents.

Dirty code

```java
public class ReportGenerator {
    // Mélange le haut niveau (génération) avec le bas niveau (manipulation de cha
    public String generateReport(List<Data> records) {
        StringBuilder report = new StringBuilder();

        // Haut niveau
        report.append("--- Rapport Mensuel ---\n");

        for (Data d : records) {
            // Bas niveau (détail d'implémentation)
            report.append("ID: ").append(d.getId());
            report.append(" | Nom: ").append(d.getName());
            report.append("\n"); // Détail de formatage de bas niveau
        }

        // Haut niveau
        report.append("--- Fin du Rapport ---");
        return report.toString();
    }
}
```

Clean code

```java
public class ReportGenerator {
    // Fonction de haut niveau : se concentre sur l'assemblage
    public String generateReport(List<Data> records) {
        StringBuilder report = new StringBuilder();

        report.append(createHeader());      // Abstraction (Haut niveau)
        report.append(createBody(records)); // Abstraction (Haut niveau)
        report.append(createFooter());      // Abstraction (Haut niveau)

        return report.toString();
    }

    // Fonction de niveau immédiatement inférieur
    private String createBody(List<Data> records) {
        StringBuilder body = new StringBuilder();
        for (Data d : records) {
            body.append(formatRecord(d)); // Abstraction (Niveau intermédiaire)
        }
        return body.toString();
    }

    // Fonction de bas niveau : se concentre sur le détail (append)
    private String formatRecord(Data d) {
        return "ID: " + d.getId() + " | Nom: " + d.getName() + "\n";
    }

    // Fonctions simples d'implémentation
    private String createHeader() { return "--- Rapport Mensuel ---\n"; }
    private String createFooter() { return "--- Fin du Rapport ---"; }
}
```

Arguments

Idéal : Le nombre idéal d'arguments pour une fonction est **zéro (niladique)**. Le minimum nécessaire est **un (monadique)**, suivi de près par **deux (dyadique)**.

Seuil Critique : Évitez les fonctions avec **trois arguments (triadique)**. Quatre arguments ou plus (polyadique) nécessitent une justification exceptionnelle, et sont généralement un signe de mauvaise conception.

Coût Conceptuel : Les arguments augmentent la **charge cognitive** (ils forcent le lecteur à interpréter le contexte à chaque appel) et **complexifient les tests** (il faut tester toutes les combinaisons).

Arguments de Sortie (Output) : Proscrivez les arguments de sortie. Le flux d'information doit être clair : les données **entrent** par les arguments et **sortent** par la valeur de retour.

Ne jamais passer un booléen (flag) en argument à une fonction. Un argument booléen est une complication immédiate du contrat de la fonction et signale de manière évidente qu'elle fait **deux choses différentes** (une si `True`, une autre si `False`).

Dirty code

```
// Très difficile à comprendre et à tester
function createAndSendReport(
    id: string,
    data: any[],
    destination: string,
    isAsync: boolean,
    results: { status: string } // Argument de sortie !
): void {

    // Le flag booléen indique que la fonction fait deux choses
    if (isAsync) {
        // Logique 1 : Tâches asynchrones
        externalAPI.sendAsync(id, data, destination);
        results.status = 'PENDING';
    } else {
        // Logique 2 : Tâches synchrones
        const report = ReportBuilder.build(data);
        FileLogger.save(report, destination);
        results.status = 'COMPLETED';
    }
}
```

Clean code

```
// Séparation des responsabilités et des arguments clairs

// Version Synchrone : 2 arguments
function createAndSaveReport(data: any[], destination: string): string {
    const report = ReportBuilder.build(data);
    FileLogger.save(report, destination);
    return 'COMPLETED'; // La sortie est le retour
}

// Version Asynchrone : 3 arguments, mais un objet comme alternative
// L'objet réduit le nombre de variables à 2 (id et config) au niveau de l'appel
function sendAsyncReport(id: string, config: { data: any[], destination: string })
    externalAPI.sendAsync(id, config.data, config.destination);
    return 'PENDING';
}

// Exemple d'appel propre (Utilisation d'un objet de configuration)
// sendAsyncReport("R001", { data: userData, destination: "/tmp/report.pdf" });
```

Effets de bord

Règle Clé : Une fonction doit soit **modifier l'état d'un objet** (si c'est son rôle), soit **retourner une valeur**, mais elle ne devrait jamais faire les deux de manière inattendue, car cela crée des **effets secondaires** cachés.

Problème Majeur : Si une fonction nommée `verifier_mot_de_passe()` non seulement vérifie le mot de passe, mais **initialise** aussi l'utilisateur ou **modifie** une variable globale, elle a un effet secondaire désinformateur.

Commutateur de Fonction : Les fonctions ne devraient pas avoir besoin d'un argument booléen pour décider si elles doivent ou non exécuter une partie de leur logique.

Dirty code

```
public class UserService
{
    private bool _isLoggedIn = false; // État global

    // Le nom suggère une requête (vérifier), mais c'est aussi une commande (modif
    public bool CheckPasswordAndLogin(string user, string pass)
    {
        bool isValid = Database.Verify(user, pass);

        if (isValid)
        {
            // Effet secondaire caché : modification d'un état global (Commande)
            _isLoggedIn = true;
            // Effet secondaire n°2 : Logique inattendue
            EmailService.SendWelcomeEmail(user);
        }

        return isValid; // Retourne une valeur (Requête)
    }
}
```

Clean code

```
public class UserService
{
    private bool _isLoggedIn = false; // État géré par des commandes claires

    // Requête : Ne fait QU'UNE CHOSE (vérifier) et retourne une valeur. AUCUN effe
    public bool IsPasswordValid(string user, string pass)
    {
        return Database.Verify(user, pass);
    }

    // Commande : Ne fait QU'UNE CHOSE (modifier l'état) et ne retourne pas de vale
    public void UserLogin(string user)
    {
        _isLoggedIn = true;
        // Commande supplémentaire est explicite
        EmailService.SendWelcomeEmail(user);
    }

    // La logique de connexion devient explicite :
    // if (IsPasswordValid(u, p)) { UserLogin(u); }
}
```

Exception ou codes d’erreur

Règle Clé : Utilisez les **exceptions** plutôt que de retourner des codes d'erreur à partir de fonctions de **commande** (fonctions qui changent l'état). Un code d'erreur viole la séparation `Command-Query` et mène à un code profondément imbriqué.

Séparer les Chemins : Les exceptions permettent de **séparer le code de traitement d'erreur** du chemin nominal ("Happy Path"), ce qui simplifie grandement la lecture et la structure.

Extraction des Blocs `Try/Catch` : Les blocs `try/catch` sont confus. Isolez-les dans une fonction dédiée. Si le mot-clé `try` est utilisé, il devrait être la première chose dans la fonction, et rien ne devrait suivre le bloc `catch/finally`.

Éviter les Aimants de Dépendance : Une énumération d'erreurs (`Error.OK, Error.INVALID`) est un **aimant à dépendances**. Toute modification de cette énumération force la recompilation et le redéploiement de nombreuses classes. Les exceptions éliminent ce problème.

DRY: Don’t repeat yourself

Règle Clé : La **duplication de code est la racine de tous les maux** en logiciel. Toute algorithme, structure ou logique répété, même légèrement masqué ou non uniforme, doit être identifié et éliminé.

Conséquences : La duplication **gonfle le code** (bloat) et multiplie par le nombre de fois où le code est copié le risque de bogues (chance d'erreur d'omission) et le **coût de maintenance** (modification N fois nécessaire).

Solution : Les innovations en génie logiciel (POO, programmation structurée, méthodes de conception) sont avant tout des stratégies pour **concentrer la logique** dans un seul endroit (base classes, fonctions, composants) et éviter la redondance.

Fabrique - patron

```python
# --- 1. La Fabrique (Factory) : Le seul endroit où se fait l'instanciation ---
class FabriqueDocument:
    @staticmethod
    def creer_document(type_demande):
        if type_demande == 'PDF':
            return DocumentPDF() # UNIQUE emplacement de la logique 1
        elif type_demande == 'HTML':
            return DocumentHTML()
        else:
            return DocumentTEXTE()

# --- 2. Les Clients : Utilisation sans connaître les détails ---
# Dans le module A
def traiter_demande(type_demande):
    document = FabriqueDocument.creer_document(type_demande)
    document.afficher() # Plus de if/else ici !

# Dans le module B
def creer_rapport(type_demande):
    rapport = FabriqueDocument.creer_document(type_demande)
    rapport.exporter() # Plus de if/else ici !
```

Commentaires

S’expliquer en code

Privilégier le nettoyage au commentaire : N'utilisez pas les commentaires pour expliquer un code désordonné ou confus ; consacrez plutôt ce temps à réusiner et à nettoyer le code lui-même.

Le code doit s'expliquer par lui-même : Un code clair et expressif nécessitant peu de commentaires est bien meilleur qu'un code complexe accompagné de nombreuses explications.

Exprimer l'intention dans le code : La meilleure documentation consiste à créer des fonctions et des noms de variables clairs qui expriment la logique, plutôt qu'à écrire des commentaires (par exemple, remplacer une condition complexe par un appel de fonction bien nommé).

Bons commentaires

Information Basique (Valeurs de Retour) : Un commentaire peut brièvement décrire ce que retourne une méthode, bien que l'idéal soit de renommer la méthode pour éliminer ce besoin (ex: responderBeingTested()).

Explication de l'Intention : Documenter pourquoi une **décision de codage** spécifique a été prise (surtout si elle est non évidente ou arbitraire), comme justifier un ordre de tri ou la mise en place d'un test de condition de concurrence (race condition).

Clarification de Valeurs Obscures : Traduire le sens d'un argument ou d'une valeur de retour peu claire, particulièrement dans le code standard ou des librairies immuables, mais cela comporte un risque élevé d'inexactitude.

Bons commentaires

Avertissements et Conséquences : Utilisez les commentaires pour alerter les autres programmeurs sur les **problèmes non évidents**, comme le manque de sécurité des threads dans une classe (SimpleDateFormat) ou la lenteur excessive d'un test.

Notes "À Faire" (//TODO) : Les commentaires **TODO** sont acceptables pour marquer les tâches futures (suppression de fonctionnalités, renommage, etc.), mais ils ne doivent jamais justifier de laisser du mauvais code dans le système.

Amplification de Détails Cruciaux : Un commentaire peut **souligner l'importance** vitale d'un détail de code qui pourrait sembler mineur (comme l'appel à une fonction .trim()) mais qui a des conséquences majeures sur la logique globale.

Mauvais commentaires

Éviter les Commentaires Énigmatiques : Ne jamais écrire de commentaire dont la signification oblige le lecteur à **chercher la réponse** dans d'autres modules ; un tel commentaire est un échec de communication.

Éliminer la Redondance : Les commentaires qui font que **lire le code prend plus de temps** que de lire le commentaire sont inutiles. Ils n'apportent aucune justification ni intention supplémentaire et ne font qu'encombrer.

Qualité sur Quantité : Écrire un commentaire uniquement parce qu'on s'y sent obligé est une mauvaise pratique. Si un commentaire est écrit, il doit être le meilleur possible et non un simple marmonnement.

Mauvais commentaires

Éviter les Mensonges Subtils : Un commentaire doit être précis ; un commentaire trompeur ou incomplet (ex: une attente masquée par un timeout) est pire que pas de commentaire, car il induit en erreur les futurs programmeurs.

Bannir le Bruit et la Redondance : Supprimez les commentaires qui répètent l'évidence (comme les Javadocs des accesseurs simples ou des constructeurs par défaut) ou qui sont imposés par des règles inutiles.

Éliminer les Journaux Historiques : Les journaux de modifications (Journal Comments) sont obsolètes grâce aux systèmes de contrôle de version modernes et doivent être entièrement retirés des fichiers sources, car ils ne font qu'encombrer le code.

Mauvais commentaires

Interdiction du Code Commenté : Ne jamais laisser de **code commenté** dans le système de production ; cela encombre et s'accumule. Utilisez plutôt le système de contrôle de version (**Git**) pour récupérer l'historique si nécessaire.

Éviter l'Information Irrévérente : Supprimez les commentaires qui contiennent des détails **historiques** ou techniques **arcaniques**, qui ne sont pas nécessaires à la compréhension immédiate du code.

La Clarté avant Tout : Un commentaire doit avoir un lien évident avec le code. Évitez les commentaires qui introduisent de nouveaux concepts obscurs ou qui nécessitent eux-mêmes une explication.

Ressources

Clean code:

https://book.northwind.ir/bookfiles/clean-code-a-handbook-of-agile-software-craftsmanship/Clean.Code.A.Handbook.of.Agile.Software.Craftsmanship.pdf

Refactoring GURU:

https://refactoring.guru/