Devoxx 2022
Pourquoi et comment refactorer

Présentation

- **Christophe Marchand**
  - cmarchand@oxiane.com
- **Développeur Java depuis 2000**
- **Formateur OXiane**
- **Développement d'outils d'automatisation des tests**
  - Pour Xslt, XQuery, Schematron
- **Software Craftmanship**

Sommaire

Sommaire

Situation

Situation

- **`Créations Merveilleuses Inc` fabrique des vistamboires**
  - Vistamboires à coins nickelés
  - Outil non identifiable, dont on a besoin, mais que l'on n'a pas !
- **Société créée, il faut maintenant vendre**
  - Process industriel Ok
  - Chaine de fabrication opérationnelle
  - Commerciaux recrutés
  - Clients trouvés
- **Comment facturer ?**
- **Produire des factures !**
  - Obligation d'avoir un outil
    - Un tableur ne suffit pas
  - L'achat d'un outil existant n'est pas envisageable

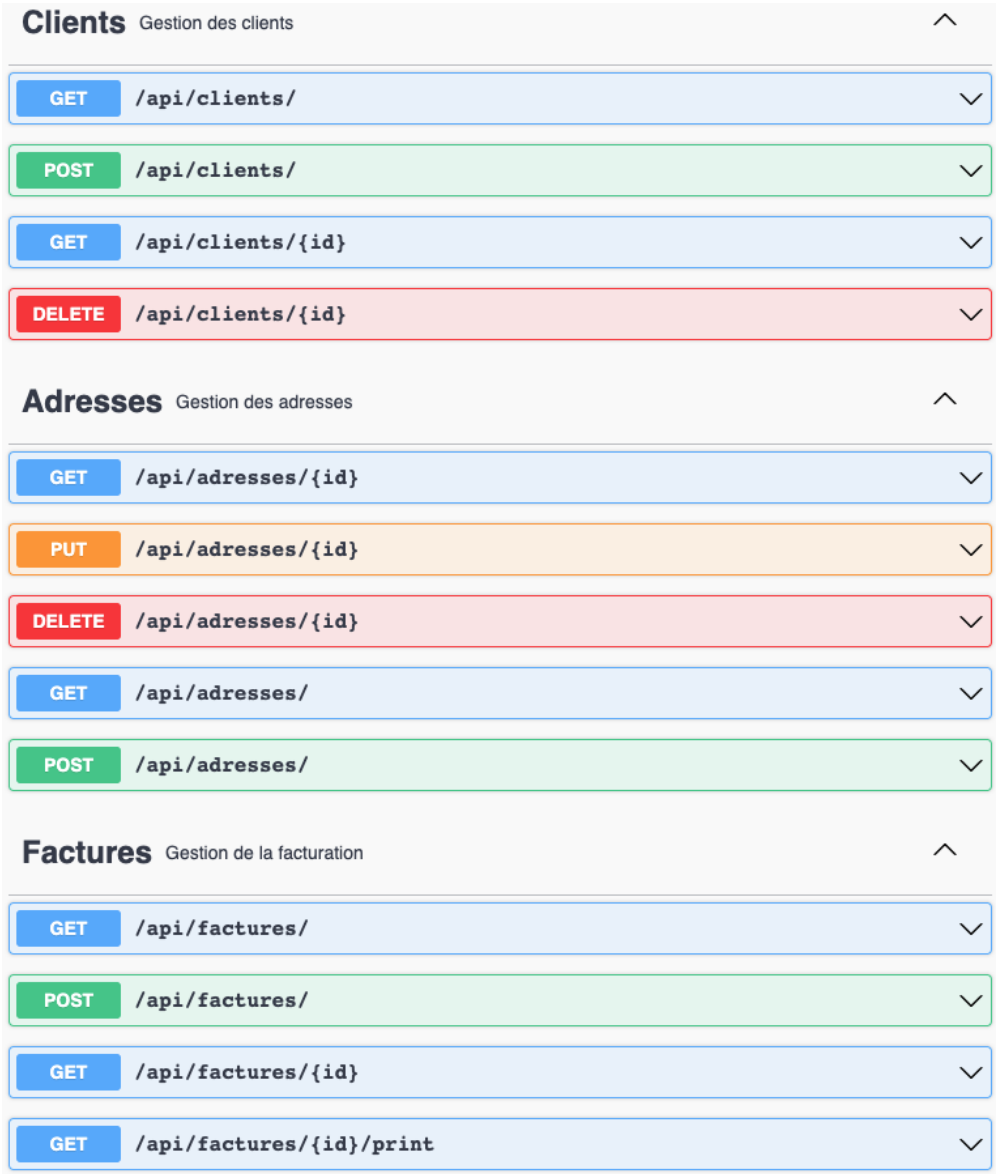
Contraintes

- **La société ne fabrique qu'un unique produit**
- **Un client veut acheter 1 vistamboire**
- **Le prix unitaire du vistamboire peut varier dans le temps**
- **Le taux de TVA du vistamboire est de 20%**
- **La facture doit respecter les règles françaises**
  - L'adresse du client doit être affichée sur la facture
  - Les montants de TVA doivent être affichés
  - Les prix unitaires doivent apparaitre
  - Les remises doivent être listées avec précision
- **La facture doit être remise au client**
  - Donc imprimée
  - Donc archivée

Externalisation

- **`Créations Merveilleuses Inc` délègue à une ESN la réalisation**
  - Choix de Java 17 / Spring Boot / Angular
  - Un développeur back Java 5 ans d'expérience
  - Un développeur Front Angular avec 4 ans d'expérience

Version 1



- **Rapidement**
  - Base de données relationnelle
    - Spring-Data / JPA
  - API Rest
    - Avec client Swagger
  - Front Angular
    - Non présenté

Démo 1

Evolutions
... plusieurs étapes ...

Evolution 1
Vendre plusieurs vistamboires !

- **Un client achète 2 vistamboires**
  - Il faut produire sa facture
  - L'équipe projet décide de gérer les cas où plus de 1 vistamboires sont achetés
- **L'équipe ajoute un attribut `quantite` dans la facture**
  - Modifie les différents services, persistance, calculs
  - Test, recette Ok
  - Livraison et mise en prod
- **Obligation de mise à jour de la base de données**
  - Ajouter une colonne **`nullable`**
  - Mise à jour de la valeur pour les factures existantes
  - Modification de la colonne en **`not null`**
  - Création d'un mécanisme générique permettant de passer des scripts SQL

Evolution 2
Particuliers et Professionnels

- **On doit facturer différemment les professionnels et les particuliers**
  - Les clients sont identifiés **`PROFESSIONNEL`** ou **`PARTICULIER`**
  - Le prix unitaire des professionnels est 30% moins cher que celui des particuliers
- **Le marketing récompense les bons clients**
  - Remise sur volume de vente annuel

Volume	Remise
> 10 / an	10 %
> 20 / an	15 %
> 50 / an	20 %

- **Il faut**
  - Modifier la structure du **`client`** et définir les existants comme **`PARTICULIER`**
  - Savoir compter le volume acheté par un client dans la dernière année
  - Revoir la méthode de calcul des montants de la **`Facture`**

Evolution 3
Prix différents par secteurs géographiques

- **Un client demande un vistamboire inoxydable**
  - L'atelier répond que le nickel est inoxydable
  - Les vistamboires à coins nickelés sont donc inoxydables
- **Le marketing veut tirer parti de cette demande spéciale**
  - Ils décident que dans les départements marins, le prix unitaire sera différent
  - Dans la facture, le libellé du vistamboire doit changer
    - **`Vistamboire inoxydable`**
  - Les remises par volume sont spécifiques à chaque secteur géographique
  - Secteur maritime

Volume	Remise
> 15 / an	10 %
> 25 / an	17 %
> 40 / an	20 %

  - Autres secteurs

Volume	Remise
> 10 / an	10 %
> 20 / an	15 %
> 50 / an	20 %

Evolution 4
Promotions

- **Un concurrent fait son entrée sur le marché**
  - Le marketing veut faire une campagne de promotion pour marquer son arrivée
- **L'équipe projet décide de construire un vrai modèle de promotions**
  - Ils savent que le marketing aura d'autres idées de promotions
  - Promotion de date à date
  - Promotion en pourcentage **ou** en montant
  - Promotion cumulable ou non à d'autres promotions
- **Si plusieurs promotions peuvent s'appliquer**
  - Si il n'y a pas de promotion exclusive
    - On applique toutes les promotions
  - Sinon
    - On sélectionne toutes les promotions exclusives
    - On sélectionne celle qui est la plus intéressante pour le client
- **Les promotions doivent apparaître sur la facture**

Démo 2

Au bout d'un an ...

Changements dans l'équipe

- **Le temps a passé, l'équipe change**
  - Le BA est parti, appelé sur un autre projet plus motivant
  - Les développeurs sont partis, remplacés par une nouvelle équipe
  - Un nouveau Talent Marketing Operation Specialist a été recruté, il a des idées

Changements dans l'équipe

- **Le temps a passé, l'équipe change**
- **Mais ...**
  - Le BA est parti, il n'a pas enregistré les docs du projet sur le réseau, c'est perdu !
  - Les nouveaux développeurs comprennent difficilement le code
  - Le TMOS voudrait créer un secteur géographique **Montagne**
    - Mais que faire des départements **Alpes Maritimes**, **Pyrénées Orientales** et **Pyrénées atlantiques** ?
  - On gère maintenant les expéditions, il faut plusieurs adresses par client
    - Adresse de livraison, adresse de facturation, adresse de chalandise
    - Comment définit-on le secteur géographique ?
  - Il faut connecter notre application à une autre, et faire des services SOAP
  - On a installé un SonarQube, et les résultats ne sont pas bons
    - 4 bugs, 1 vulnerability
    - 44 code smells, 37% de code coverage

Analyse de la base de code

Analyse de la base de code

- **Les nouveaux développeurs ont du mal à entrer dans le code**
  - Il sont pourtant expérimentés, motivés et travailleurs
  - La documentation ayant disparu, cela n'aide pas
    - Le code en lui même n'est pas parlant
- **Les prochaines interventions sur le code seront complexes**
  - Il faudra passer beaucoup de temps à comprendre comment ça fonctionne
  - Ca risque de coûter cher

Analyse de la base de code

- **Les nouveaux développeurs ont du mal à entrer dans le code**
  - Il sont pourtant expérimentés, motivés et travailleurs
  - La documentation ayant disparu, cela n'aide pas
    - Le code en lui même n'est pas parlant
- **Les prochaines interventions sur le code seront complexes**
  - Il faudra passer beaucoup de temps à comprendre comment ça fonctionne
  - Ca risque de coûter cher
- **Le code manque de lisibilité**

Analyse de la base de code

- **Ajouter la dernière fonctionnalité a impacté beaucoup de code**
  - 4 classes créées
    - **`Promotion`** : nouvelle classe
    - **`PromotionRepository`** : nouvelle interface
    - **`PromotionRepositoryCustom`** : nouvelle interface
    - **`PromotionRepositoryImpl`** : nouvelle classe
  - 4 classes modifiées
    - **`FacturePrinterImpl`**
    - **`Adresse`**
    - **`Facture`**
    - **`FactureController`**
- **Evolution adresses et secteurs géo : impact probablement fort**
  - Evolution du modèle
  - Evolution des règles de calcul
  - Evolution de l'impression

Analyse de la base de code

- **Ajouter des service SOAP va entrainer de la duplication de code**
  - Le code métier est dans la classe `FactureController`
    - C'est la classe des services REST de la facture
- **Ou des mélanges de genre**
  - Service SOAP et Service REST : même classe ?

Analyse de la base de code

- **Ajouter des service SOAP va entrainer de la duplication de code**
  - Le code métier est dans la classe `FactureController`
    - C'est la classe des services REST de la facture
- **Ou des mélanges de genre**
  - Service SOAP et Service REST : même classe ?
- **Le code est rigide**

Analyse de la base de code

- **Les métiers, vu les changements, sont inquiets**
  - Changements dans l'équipe projet
  - Changements fonctionnels à venir importants
- **Vu la complexité, on craint qu'il y ait des régressions**
  - Est-ce que ce qui marchait et ne doit pas être modifié fonctionnera toujours ?

Analyse de la base de code

- **Les métiers, vu les changements, sont inquiets**
  - Changements dans l'équipe projet
  - Changements fonctionnels à venir importants
- **Vu la complexité, on craint qu'il y ait des régressions**
  - Est-ce que ce qui marchait et ne doit pas être modifié fonctionnera toujours ?
- **Le code semble être fragile**

Analyse de la base de code

- **Peut-on lancer un chantier d'évolutions dans ces conditions ?**

Analyse de la base de code

- **Peut-on lancer un chantier d'évolutions dans ces conditions ?**
- **Pas en toute sécurité**

Solutions ?

Solutions ?
Réécrire ?

- **Réécrire le code**
  - Pour qu'il soit plus lisible
  - Pour qu'il soit plus robuste
  - Pour qu'il soit plus souple

Solutions ?
Réécrire ?

- **Réécrire le code**
  - Pour qu'il soit plus lisible
  - Pour qu'il soit plus robuste
  - Pour qu'il soit plus souple
- **Personne n'a les moyens de réécrire du code**
  - Coût important
  - Délais importants
  - Risques importants
  - Comment faire cohabiter la version en prod et la version en cours de réécriture ?
- **Réécrire le code est un aveu d'échec**

Solutions ?
Réécrire : non

- **Réécrire le code**
  - Pour qu'il soit plus lisible
  - Pour qu'il soit plus robuste
  - Pour qu'il soit plus souple
- **Personne n'a les moyens de réécrire du code**
  - Coût important
  - Délais importants
  - Risques importants
  - Comment faire cohabiter la version en prod et la version en cour de réécriture ?
- **Réécrire le code est un aveu d'échec**
- **Il est impossible de réécrire un projet**

Solutions ?
Refactorer !

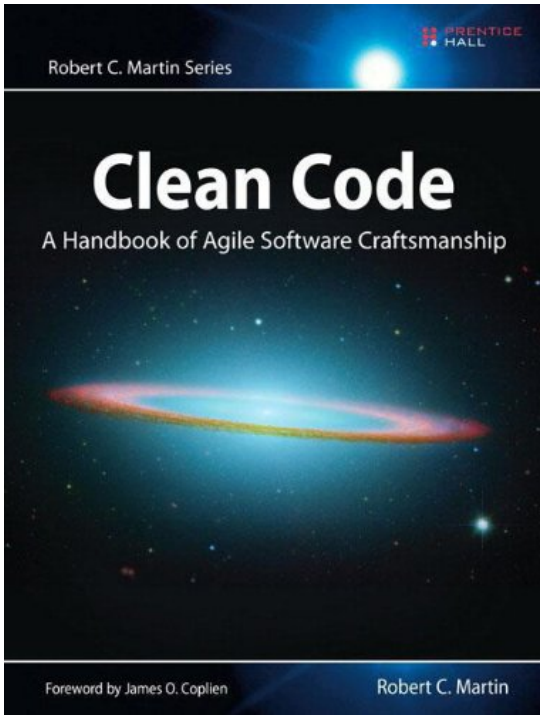
- **Faire évoluer la base de code**
- **Sans changer son comportement**
  - Aucune nouvelle fonctionnalité
  - Aucune régression
- **Le refactoring est une pratique ancienne**
  - Née avec le TDD et l'eXtreme Programming dans les années 1995
- **Objectifs**
  - Rendre le code lisible
  - Rendre le code robuste
  - Rendre le code maintenable
- **Avec quelles règles ?**
  - Respecter les principes du Clean Code
  - Respecter les principes SOLID
  - Respecter les principes de l'eXtreme Programming

Refactorer
Principes

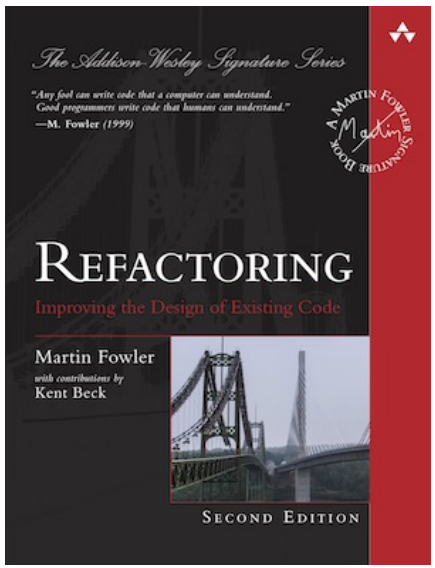
- **On ne doit pas changer le comportement du code**
- **Le code qui va être modifié doit être intégralement testé**
  - Le seul moyen de prouver ce que fait le code
- **Utiliser des outils de code coverage**
    - Pour vérifier que tout est couvert
- **Utiliser des outils de mutation testing**
  - Pour détecter des choses qui ne seraient pas correctement testées
- **Ne jamais refactorer du code non testé**
- **Vérifier que chaque opération de refactoring ne change rien**
  - Lancer les tests unitaires systématiquement
    - Ils doivent s'exécuter rapidement

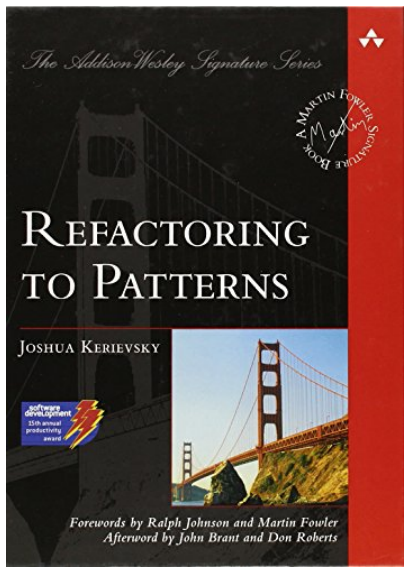
Refactorer
Principes

- **`Clean Code`**
  - Livre de Bob Martin
  - Explique ce qu'est du bon code
    - Et ce qui n'est pas du bon code
  - Montre des techniques pour conserver son code propre
- **Plusieurs principes simples**
  - Nommage explicite
  - Fonctions courtes avec peu de paramètres
  - Pas de commentaires
  - Etc...
  - ISBN : 978-0-13-235088-4



- **Refactorer ne s'invente pas, il faut apprendre et s'entrainer**
  - Lire des livres
  - Pratiquer des katas
- **Martin Fowler : `Refactoring`**
  - ISBN : 978-0-13-475759-9
- **Joshua Kerievsky : `Refactoring to patterns`**
  - ISBN : 978-0-321-21335-8





Refactorer...
pour améliorer la lisibilité

- **On passe 10 fois plus de temps à lire du code qu'à écrire du code**
  - Bob Martin, Clean Code

Refactorer...
pour améliorer la lisibilité

- **On passe 10 fois plus de temps à lire du code qu'à écrire du code**
  - Bob Martin, Clean Code
- **Si on passe 4 fois plus de temps à écrire du code**
- **Pour le rendre 2 fois plus lisible**

Refactorer... pour améliorer la lisibilité

- **On passe 10 fois plus de temps à lire du code qu'à écrire du code**
  - Bob Martin, Clean Code
- **Si on passe 4 fois plus de temps à écrire du code**
- **Pour le rendre 2 fois plus lisible**
- **On diminue le coût de 20 %**
  - 1 heure de code + 10 heures de lecture = 11h
  - 4 heures de code + 5 heures de lecture = 9h
  - Gain : 2 heures
- **Après 1 heure de code**
  - Prendre 3 heures à rendre mon code plus lisible
  - Economie de 20% sur le coût de revient du code

Refactorer...
pour améliorer la lisibilité

- **20 étapes de refactoring successives**
- **6 méthodes de refactoring utilisées**
  - **`Extract Method`** - Refactoring page 106
  - **`Slide Statements`** - Refactoring page 223
  - **`Inline Variable`** - Refactoring page 123
  - **`Extract Variable`** - Refactoring page 119
  - **`Change Signature`** - Refactoring page 124
  - **`Suppression commentaire`** - Clean Code, Redundant Comment, page 60
- **Nommage précis**
  - Chaque variable a un nom qui exprime sans doute possible ce que c'est
  - Chaque méthode a un nom qui exprime sans doute possible ce qu'elle fait
- **Méthodes courtes**
  - Facile de comprendre le code qui s'y trouve

Refactorer...
pour améliorer la lisibilité

- **Lecture de haut en bas**
  - En partant de la méthode, on lit quelques lignes
  - Les noms des instructions indiquent clairement les objectifs métier
  - Si on veut plus de détails, on continue la lecture pour approfondir
- **On gagne en rapidité de lecture et de compréhension**
- **Ce travail permettra d'économiser du temps dans le futur**
  - Et donc d'avoir un coût de maintenance/évolution plus faible

Refactorer...
pour améliorer la lisibilité

- **Lecture de haut en bas**
  - En partant de la méthode, on lit quelques lignes
  - Les noms des instructions indiquent clairement les objectifs métier
  - Si on veut plus de détails, on continue la lecture pour approfondir
- **On gagne en rapidité de lecture et de compréhension**
- **Ce travail permettra d'économiser du temps dans le futur**
  - Et donc d'avoir un coût de maintenance/évolution plus faible
- **Rendre le code lisible fait économiser de l'argent**

Refactorer... pour respecter les principes

- **5 principes solides**
  - Enoncés par Bob Martin
  - **S** : **S**ingle Responsability Principle
  - **O** : **O**pen / Closed Principle
  - **L** : **L**iskov Substitution Principle
  - **I** : **I**nterface Segregation
  - **D** : **D**ependency Inversion
- **3 principes eXtreme Programming**
  - **DRY** : **D**on't **R**epeat **Y**ourself
  - **YAGNI** : **Y**ou **A**in't **G**onna **N**eed **I**t
  - **KISS** : **K**eep **I**t **S**tupid **S**imple

Refactorer...
pour respecter les principes

- **Single Responsability Principle**
  - On ne doit avoir qu'une seule raison de modifier un bout de code
- **Si il existe plusieurs raison de modifier un code**
  - Il faut regrouper ce qui est modifié pour des raisons identiques
  - Il faut séparer ce qui est modifié pour des raisons différentes
  - Il faut séparer ce qui évolue dans des échelles de temps différentes
- **Valable pour**
  - Un groupe de lignes de code
  - Une méthode / fonction
  - Une classe / un objet
  - Un module

Robert C. Martin

Refactorer...
pour respecter les principes

- **Open / Closed Principle**
  - Fermé à la modification
  - Ouvert à l'extension
- **Quand j'ai un nouveau cas de gestion**
  - Je ne dois pas modifier de code existant
  - Je dois simplement ajouter du code
    - Par extension
    - Par composition (de préférence)
- **Souvent résolu par un pattern `Strategy`**

Bertrand Meyer

Refactorer...
pour respecter les principes

- **Liskov Substitution Principle**
  - On doit pouvoir substituer à une classe
  - N'importe laquelle de ses descendantes
  - Sans compromettre le fonctionnement du code
- **Une sous-classe doit respecter le contrat**

Barbara Liskov

Refactorer... pour respecter les principes

- **Interface Segregation Principle (Introduit par Bob Martin)**
  - Aucun code ne doit dépendre de méthodes qu'il n'utilise pas
- **Une classe doit utiliser toutes les méthodes de l'interface référencée**
  - Les interfaces doivent comporter peu de méthodes
  - Les interfaces sont déclarées côté client
  - Une même classe de service peut implémenter plusieurs interfaces

Refactorer...
pour respecter les principes

- **Dependency Inversion Principle (Introduit par Bob Martin)**
  - Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau
  - Les abstractions ne doivent pas dépendre des détails
  - Les détails doivent dépendre des abstractions
- **Permet de limiter les dépendances de compilation**
  - Et donc simplifie les livraisons
- **Principe à la base de l'Inversion of Control**
  - Mis en œuvre par Spring, entre autres

Refactorer...
pour respecter les principes

- **Don't Repeat Yourself**
  - Pas de duplication de code
  - SonarQube donne ces informations
- **Attention à ne pas tomber dans l'excès inverse**
  - Ces 2 blocs de code sont différents

```
public void sendFactureViaEMail() {
  Facture facture = getFacture();
  sendEMailToClient(facture);
  save(facture);
}
```

```
public void sendFactureViaPost() {
  Facture facture = getFacture();
  sendPostalMailToClient(facture);
  save(facture);
}
```

  - Ils ne doivent pas être mutualisés
    - Violation du Single Responsability Principle

Refactorer...
pour respecter les principes

- **You Ain't Gonna Need It**
  - Ne pas écrire de choses inutiles
  - Ne pas écrire de code réutilisable
    - Le code n'est jamais réutilisé
  - Ecrire du code jetable
    - Le jour où le code ne convient plus, on le jette et on le remplace par un autre
- **La sur-ingénierie coûte très cher**
  - Complexification de la base de code
  - Difficulté de maintenance
    - Les objectifs métier du code ne sont pas compréhensibles

Refactorer... pour respecter les principes

- **Keep It Stupid Simple**
  - Conserver des choses simples
  - Même si on pense qu'on peut factoriser des choses
    - Penser à respecter les principes SOLID dans la factorisation
- **Ne pas utiliser des syntaxes extra-ordinaires**
  - Penser que tous les développeurs n'ont pas la même aisance

```
value ^= masque;
```

- **Ne pas utiliser des constructions complexes sans raison**

```
public abstract class AbstractSingletonStubFactoryBean {
  abstract String getValue();
}
```

Démo 4

Refactorer... pour respecter l'Open Closed Principle

- **Open Closed Principle**
  - On ne doit pas modifier de code existant pour ajouter des cas de gestion
- **Ce code ne respecte pas l'Open Closed Principle**
  - Ni le Single Responsability Principle, d'ailleurs !

```
@Override
public BigDecimal calculatePrixUnit(Vistamboire vistamboire, Client client) {
  // étape 1 : en fonction du type de client
  BigDecimal prixUnitaireTypeClient;
  if (Client.TYPE_PARTICULIER.equals(client.getType())) {
    prixUnitaireTypeClient = vistamboire.getPrixUnitaireHT();
  } else if (Client.TYPE_PROFESSIONNEL.equals(client.getType())) {
    prixUnitaireTypeClient = vistamboire.getPrixUnitaireHT().multiply(new BigDecimal(0.7));
  } else {
    prixUnitaireTypeClient = vistamboire.getPrixUnitaireHT();
  }
  // étape 2 : en fonction du secteur géographique
...
}
```

Démo 5

Refactoring
Bilan

Refactoring - Bilan
Pour les développeurs

- **Un code plus lisible**
  - Montre les intentions métier
    - Qu'on avait perdu !
  - Plus facile à comprendre
- **Intégration des nouveaux développeurs plus rapide**
  - En lisant le code, ils retrouvent les concepts abordés à la machine à café
  - L'implémentation des fonctionnalités est exprimée en termes métier
    - Plus facile de discuter d'algorithmie avec les métiers

- **Le code est moins fragile**
  - La modification d'une règle de gestion n'entraîne pas de risque de régression
  - L'ajout d'un cas de gestion à une règle n'entraine pas de risque de régression

- **Le code est moins rigide**
  - Chaque méthode / classe n'a qu'une seule raison d'être modifiée
  - La modification d'une règle ne devrait pas impacter beaucoup de code
    - Ajout d'un nouveau type de client
    - Changement des bornes quantitatives pour les remises par quantité
  - L'ajout d'une nouvelle fonctionnalité ne modifiera pas de code existant
    - Services SOAP
    - Merci Spring pour l'**`@Autowired`** !

Refactoring - Bilan
Pour les métiers

- **Les corrections de bug coûtent moins cher**
  - En fait, il y a même beaucoup moins de bugs !
    - Il y a beaucoup plus de tests unitaires
  - Le temps d'identification du code responsable est plus court
  - Le temps de correction est plus court
  - Il n'y a plus de régression
- **Les évolutions coûtent moins cher**
  - Peu de code à modifier
  - Pas de risque de régression
- **Le code s'est simplifié**
- **Confiance accrue entre métier et dévs**
  - Quand on demande quelque chose, la réponse est toujours **Bien sûr !**

Résumé

Résumé

- **Le code ne naît pas complexe**
  - Il devient complexe au fur et à mesure des évolutions
- **On refactorise pour rendre le code maintenable**
  - Moins cher à faire évoluer
  - Pour économiser de l'argent dans le futur
- **On ne refactorise que du code testé**
  - Et on s'assure de la bonne qualité des tests (couverture, mutation)
- **On applique les opérations de refactoring connues**
  - Voir les livres Refactoring et Refactoring to Patterns
  - On s'appuie sur les capacités de son IDE
  - On évite d'inventer de nouvelles méthodes de refactoring
- **On s'entraîne au refactoring**
  - Au travers de katas qu'on pratique souvent

Ressources

Ressources

- **Livres**
  - Clean Code - Robert C. Martin - 978-0-13-235088-4
  - Refactoring - Martin Fowler - 978-0-13-475759-9
  - Refactoring to Patterns - Joshua Kerievsky - 978-0-321-21335-8
- **Cette présentation**
  - https://github.com/cmarchand/devoxx2022

Questions ?