INF2015 – Développement de logiciels dans un environnement Agile

Refactoring

Jacques Berger

Objectifs

Introduire une pratique de base des méthodes Agiles

Prérequis

Java

Refactoring

Réécrire
Retravailler
Refaire le design
Refaire l'architecture

Modifier du code existant, sans modifier la fonctionnalité

Pourquoi?

Rendre le code plus lisible, plus clair

Faciliter le changement

Entretenir le système

Pourquoi?

Le code tend à pourrir avec le temps

Plus on ajoute de la fonctionnalité, plus le code devient difficile à lire et à comprendre

Pourquoi?

Faire du refactoring dans un programme, c'est comme enlever les mauvaises herbes dans un jardin

Il faut le faire tôt, il faut le faire souvent, continuellement

Raffinement successif

Les méthodes Agiles font la promotion du raffinement successif :

Faire du refactoring continuellement durant le développement du système

Si on doit travailler dans une classe, on la nettoie un peu avant pour faciliter notre travail

Quand?

Le code est mal placé :
    Variables, méthodes, classes

Les objets sont mal nommés

Duplication de code – DRY
    Don't Repeat Yourself

Quand?

Design non orthogonal

- Un changement dans une classe entraîne un changement dans une autre classe

Connaissances dépassées

- On connait mieux le problème aujourd'hui

Performance

- Déplacer le code pour améliorer les performances

Quand?

Conclusion

Tout ce qui semble mal doit être amélioré

Problèmes

La peur de toucher à ce qui fonctionne déjà

La réaction des patrons
«Ce code fonctionne mais je dois le réécrire»

Tendance : on fait le refactoring à la fin du projet
Réalité : on manque de temps, on coupe le refactoring

Problèmes

Dette technique
- En garder une trace
- Inutile si le refactoring est continuel

Comment?

Activité qu'il faut réaliser tranquillement, de façon délibérée et avec soin

- On ne veut pas introduire de nouveaux bogues

Ne pas faire de refactoring et ajouter de la fonctionnalité en même temps

Comment?

Découper le refactoring en plusieurs petites étapes distinctes

- Déplacer un champ
- Fusionner 2 méthodes
- Déplacer une méthode

Faire autant de petites étapes que possible et tester après chaque petite modification

Changement majeur

Lorsqu'on modifie une méthode qui possède beaucoup d'appelants et qu'on doit réviser chaque appelant :

Modifier la fonction pour que les appelants ne compilent plus, ça évite les oublis

Bogue

Lorsqu'on trouve un nouveau bogue pendant une activité de refactoring, que faire?

On laisse le bogue en place, certains appelants peuvent profiter de ce bogue sans le savoir, le corriger pourrait provoquer des erreurs inattendues

Bogue

On documente le bogue trouvé et on termine le refactoring, nous reviendrons pour corriger ce bogue plus tard

Les nouveaux bogues potentiels dans les appelants seront plus difficiles à détecter, vaut mieux faire le refactoring et la correction séparément

Tests

Toute manipulation de code peut introduire des erreurs dans le logiciel

- Le refactoring ne fait pas exception

La présence de tests unitaires favorise le refactoring

Tests

Les tests unitaires permettent de vérifier très rapidement si un refactoring a brisé la fonctionnalité en place

Ils favorisent le débogage et donnent confiance au développeur

Tests

Il est recommandé d'avoir une bonne couverture de tests avant de faire du refactoring

IDE

Les IDE modernes fournissent des fonctionnalités de refactoring automatiques

- Renommer des objets, des variables, etc.

Cas

Observons quelques cas...