Refactoring

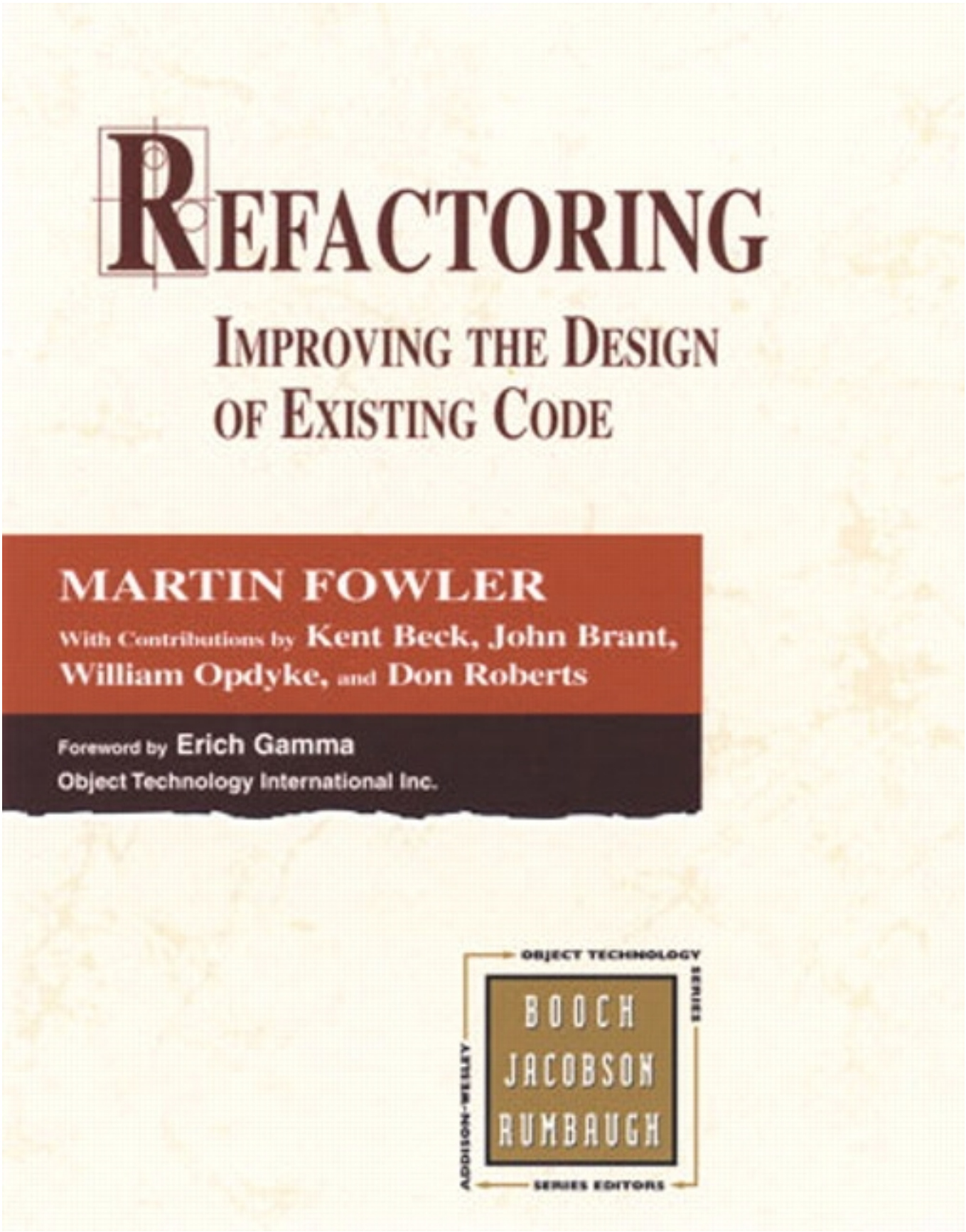
par Matthieu Brucher (http://matthieu-brucher.developpez.com/) (Blog)

Date de publication : 15/05/2006

Dernière mise à jour : 28/08/2006

Critique du livre Refactoring de *Martin Fowler*

I - Description



Pendant que l'application de technologies objet - particulièrement dans le langage de programmation Java - devient courant, un nouveau problème est apparu dans la communauté du développement logiciel. Un nombre significatif de programmes mal designés ont été créés par des développeurs moins expérimentés, se traduisant par des applications qui sont inefficaces et difficiles à maintenir et à étendre. Les professionnels des systèmes logiciels découvrent de plus en plus la difficulté de travailler avec ces applications héritées, "non optimales". Pendant plusieurs années, les programmeurs objets d'un niveau d'expert ont employés une collection grandissante de techniques pour améliorer l'intégrité et la performance de ce genre de programmes existants. Connus sous le nom de "refactoring", ces pratiques sont restées dans le domaine des experts car aucune tentative avait été menée pour les transcrire dans une forme que tous les développeurs pouvaient utiliser... jusqu'à aujourd'hui. Dans Refactoring: Improving the Design of Existing Code, Martin Fowler, mentor renommé de la technologie objet ouvre une nouvelle voie, démystifiant ces pratiques avancées et démontrant comment les développeurs peuvent se rendre compte des bénéfices de ce nouveau processus.

Avec un entraînement adéquat, un designer aguerri de système peut prendre un mauvais design et le transformer en code robuste, bien designé. Dans ce livre, Martin Fowler nous montre où sont traditionnellement les opportunités pour le refactoring et comment passer d'un mauvais design à un bon. Chaque étape de refactoring est simple - apparemment trop simple pour mériter d'être effectuée. Refactorer peut impliquer déplacer un champ d'une classe à une autre, extraire du code d'une méthode pour en faire une nouvelle ou même modifier une hierarchie. Si ces étapes individuelles peuvent sembler élémentaires, l'effet cumulatif de ces petits changements peut améliorer grandement un design. Refactorer est un moyen efficace pour empêcher un logiciel de périclitter.

Outre les discussions sur les différentes techniques de refactoring, l'auteur donne un catalogue détaillé de plus de 70 étapes avec des références judicieuses pour nous apprendre quand les appliquer, des instructions pas à pas pour chaque application d'un tel pattern et un exemple illustrant comment fonctionne le refactoring. Les exemples illustratifs sont écrits en Java, mais les idées sont applicables à tout langage orienté objet.

II - Table des matières

- 9. Simplifying Conditional Expressions.
  - Decompose Conditional.
  - Consolidate Conditional Expression.
  - Consolidate Duplicate Conditional Fragments.
  - Remove Control Flag.
  - Replace Nested Conditional with Guard Clauses.
  - Replace Conditional with Polymorphism.
  - Introduce Null Object.
  - Introduce Assertion.
- 10. Making Method Calls Simpler.
  - Rename Method.
  - Add Parameter.
  - Remove Parameter.
  - Separate Query from Modifier.
  - Parameterize Method.
  - Replace Parameter with Explicit Methods.
  - Preserve Whole Object.
  - Replace Parameter with Method.
  - Introduce Parameter Object.
  - Remove Setting Method.
  - Hide Method.
  - Replace Constructor with Factory Method.
  - Encapsulate Downcast.
  - Replace Error Code with Exception.
  - Replace Exception with Test.
- 11. Dealing with Generalization.
  - Pull Up Field.
  - Pull Up Method.
  - Pull Up Constructor Body.
  - Push Down Method.
  - Push Down Field.
  - Extract Subclass.
  - Extract Superclass.
  - Extract Interface.
  - Collapse Hierarchy.
  - Form Template Method.
  - Replace Inheritance with Delegation.
  - Replace Delegation with Inheritance.
- 12. Big Refactorings.
  - Tease Apart Inheritance.
  - Convert Procedural Design to Objects.
  - Separate Domain from Presentation.

III - Critique : le Refactoring expliqué simplement

Une référence, voilà ce qu'est devenu ce livre pour moi. Le catalogue de patterns - parce que je considère ces petites étapes comme des patterns - est dense, presque exhaustif - je suis sûr qu'on pourrait en trouver d'autres -, mais il y a les patterns les plus simples et les plus utiles. Comme le texte au dos du livre le dit, certaines étapes semblent vraiment trop simples, et c'est le cas. Tout le monde sait inliner une fonction. On a un peu plus de mal en extrayant une méthode d'une autre méthode, mais de manière générale, on serait tomber tôt ou tard sur ces patterns.

L'avantage du livre, ce n'est pas de nous redire ce qu'on sait, c'est de nous montrer une étape à laquelle on n'a pas pensé et qui résoud le problème qu'on a. Enfin, c'est pas un problème, c'est juste que le code commence à sentir mauvais. D'ailleurs, l'odeur du code, c'est une des caractéristiques de ce livre, *Martin Fowler* nous donne des pistes pour apprendre à reconnaître l'odeur d'un code. Ca m'a fait sourire de se dire que le code avait une odeur. Et c'est vrai que lorsqu'on regarde le code qu'on faisait quand on était plus jeune, on se rend compte qu'il pue.

Un autre point de ce livre, c'est qu'il met l'accent sur les tests, principalement unitaires. Et effectivement, pour un livre qui parle de refactoring, c'est important. Il faut bien que le code après modification ait le même effet que sans modification ! L'exemple qui est donné est aussi explicite, on utilise plusieurs méthodes au fur et à mesure de l'avancement de la procédure, c'est très intéressant de voir comment ça marche.

Enfin, le code est du Java, mais pour quelqu'un qui connait le C++, ça passe sans problème. Chaque opération est suffisemment simple pour être presque identique à son équivalent en C++. Enfin, un petit topo sur ce qui existait à l'époque sur le refactoring est donné, et même s'il est un peu dépassé, ce topo est très instructif.

IV - Liens annexes

Critique sur la page de livres Conception

Achat sur Amazon.fr

Lien vers le site de l'éditeur