Software Reengineering:
Refactoring Assistants

Henrique Rocha

Universiteit Antwerpen

Refactoring: change the internal structure of a code without compromising its external behavior [FBB+99].

Introduction

“I wrote the original edition in 2000 when Refactoring was a little-known technique.” – Martin Fowler

- Currently, refactoring is a well-known concept for developers in industry and academia alike.
- In academia, we have been discussing and studying refactoring for decades.

Strategic Refactoring

- Strategic Refactoring is to apply refactoring for a particular design reason/goal.
  - Support a new feature/correction
  - Solving a specific design problem
  - “Refactor to Understand” (OORP, p.127)

- In this Reengineering Course, refactoring without a reason/goal is meaningless.
- Please remember the pattern “Keep it Simple” (OORP, p.37) when planning refactoring activities.

Bad/Code Smells

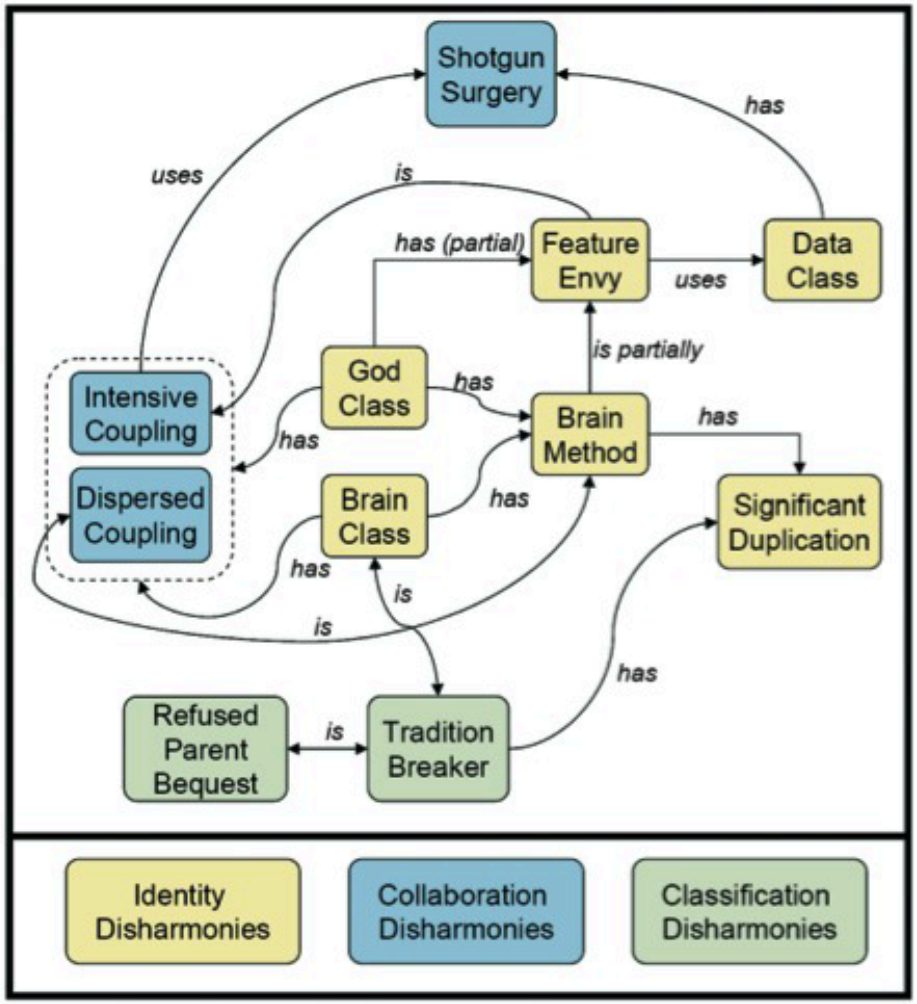


Fig. 4.12. Disharmonies and their correlations.

Code Smell is a poorly designed piece of code.

Code smells are prime candidates for refactoring.

SonarQube is a nice tool for Smell detection.

In CodeScene, only the paid version shows Smells

Code Smells Example: Feature Envy

- This code smells occurs when a method that belongs to a class, uses more methods/attributes from another.
- In other words, the method is envious of the features provided by the other class as it is using them more often.

- Refactoring solution: Move Method
  - If method is using more features from another class, then maybe that class is a more appropriate place for it.

Code Smell Example: God Class

- A God Class is a class that is big on size and/or responsibilities, controlling too many objects.

- Refactoring solution: Extract/Split Class
  - It is often possible to “split” a god class into two or more classes with a more clear and logical design

Guidelines on How to Refactor

(1) Identify where (and when) to refactor

(2) Consider which refactoring(s) to apply

(3) Assure behavior preservation on the refactored artifact

(4) Perform the refactoring(s)

(5) Assess the effect of the refactoring on quality

(6) Maintain the system's consistency among the refactored code and other software artifacts

(1) Where to Refactor

- “Most Valuable First” (OORP, p.29)
- Several tools can help you find artifacts that may need refactoring (e.g., CodeScene)
- Metrics & Visualization approaches can show you artifacts with jarring characteristics (Study the Exception Entities, OORP, p.107)
- Code/Design Smells can also be good indicators.
- Dependency analysis may show coupled/entangled artifacts.

(2) Consider Which Refactorings

- There are several types of refactorings we can apply to source code artifacts.
- We have catalogues defining these refactorings.
- The definitions are important as it provides a common terminology (e.g., Move Method, Extract Method)

(3) Preserving the Behavior after Refactoring

- Nothing better than a good set of test cases.
- Refactoring tools can assure (to some degree) that the refactor activity will not break the artifact (it does not for the whole system… only the artifact).
- Perform refactor in small steps
- Collaborative code management systems for safety (version control, separate branches)

(4) Perform the Refactoring

- Most IDEs can perform some refactoring operations
- Dedicated tools usually perform better (JDeodorant, JMove, etc).
- You can also do it manually (complex refactorings).
- It is a good practice to try to limit your refactorings per commit.

- In the **Final Project**, each refactoring activity should be on a separate commit (and the commit message should indicate the what was done).

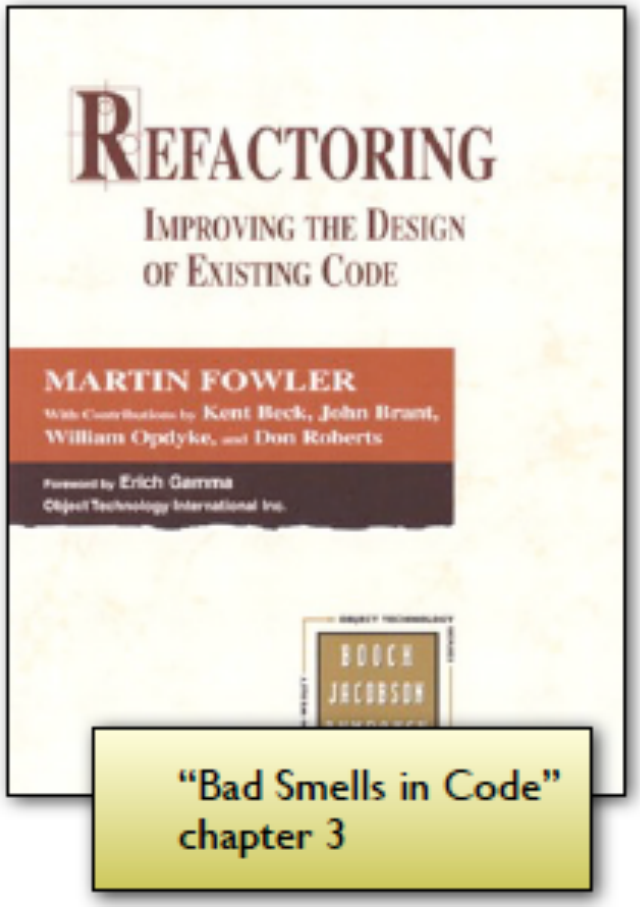
(5) Assessing the Effects on Quality

- Re calculate your analysis (tools) and verify the new structure
- Metrics (Complexity), Visualizations, Dependencies, Smells detectors, …

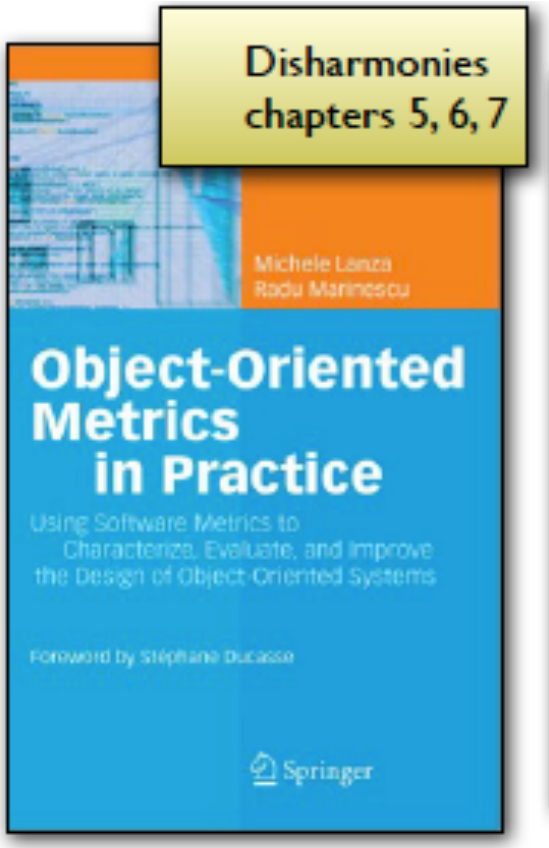
(6) Maintain the System's Consistency

- After the refactoring it is important to verify the system's consistency.
- First, a system-level test to assure you did not break another part of the system
- Then, don't forget about maintaining the consistency on other non-code artifacts
  - Documentation (specially design documents like UML diagrams), API interfaces, test cases.

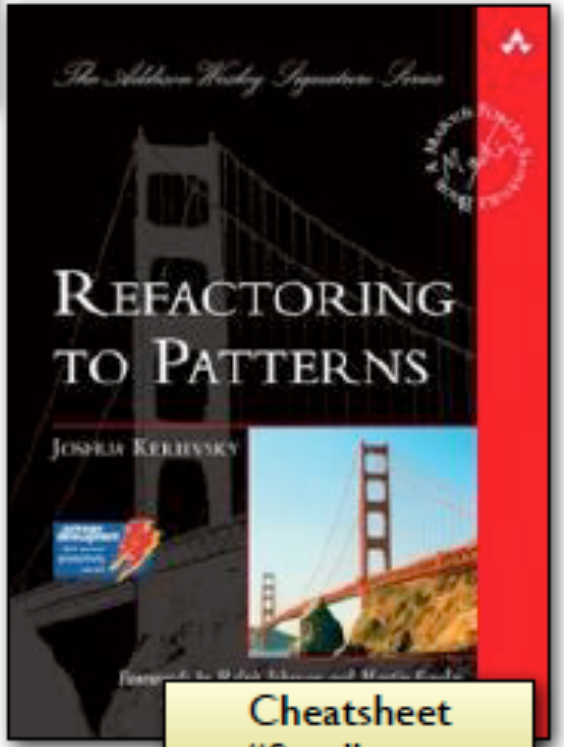
Refactoring Books



"Bad Smells in Code" chapter 3



Disharmonies chapters 5, 6, 7



Cheatsheet "Smells to Refactorings"

Refactoring Catalogues

- The books from the last slide have refactoring catalogues listed on then.
- The course book also has a small refactoring catalogue (OORP, p.317)
- Martin Fowler maintains an online catalogue https://refactoring.com/catalog/
- Refactoring Guru https://refactoring.guru/refactoring/catalog

Refactoring for the Project

- Intermediate Report
  - Describe which tools, techniques, and patterns (from OORP) were used to identify refactoring targets
  - Reason why those refactoring are important for your goal
  - Describe the planned refactorings activities

- Final Report
  - Same as Intermediate Report, but the refactorings must be “completed” by now
  - Each refactoring should be on its own commit in the GitHub repository