Refactoring To Patterns

What Is Refactoring To Patterns?

Refactoring to Patterns is the marriage of refactoring -- the process of improving the design of existing code -- with patterns, the classic solutions to recurring design problems. Refactoring to Patterns suggests that using patterns to improve an existing design is better than using patterns early in a new design. This is true whether code is years old or minutes old. We improve designs with patterns by applying sequences of low-level design transformations, known as refactorings.

What Are The Goals Of This Book?

This book was written to help you

- Understand how to combine refactoring and patterns
- Improve the design of existing code with pattern-directed refactorings
- Identify areas of code in need of pattern-directed refactoring
- Learn why using patterns to improve existing code is better than using patterns early in a new design

To achieve these goals, this book features

- A catalog of 27 refactorings
- Examples based on real-world code, not the toy stuff
- Pattern descriptions, including real-world pattern examples
- A collection of smells (i.e. problems) that indicate the need for pattern-directed refactorings
- Examples of different ways to implement the same pattern
- Advice for when to refactor to, towards or away from patterns
- A suggested study sequence for the refactorings

Why Refactor?

"Grow, don't build software"

— Fred Brooks

> The reality:

— Extremely difficult to get the design "right" the first time

— Hard to fully understand the problem domain

— Hard to understand user requirements, even if the user does!

— Hard to know how the system will evolve in five years

— Original design is often inadequate

— System becomes brittle over time, and more difficult to change

> Refactoring helps you to

— Manipulate code in a safe environment (behavior preserving)

— Recreate a situation where evolution is possible

— Understand existing code

When to refactor?

- When is it best for a team to refactor their code?
  - best done **continuously** (like testing) as part of the process
  - hard to do well late in a project (like testing)
  - Refactor when you identify an area of your system that:
  - isn't well designed
  - isn't thoroughly tested, but seems to work so far
  - now needs new features to be added

Who Is This Book For?

This book is for object-oriented programmers engaged in or interested in improving the design of existing code. Such programmers either

- Use patterns and/or practice refactoring, but have never implemented patterns by refactoring
- Know little about refactoring and patterns and would like to learn more

The types of projects for which this book is useful are:

- Greenfield development, in which a new system or feature is being written from scratch
- Legacy development, in which you are mostly maintaining a legacy system.

Whether you are doing greenfield or legacy development, the refactorings and design ideas in this book will help you with your work.

What Background Do You Need?

This book assumes you are familiar with design concepts like tight- or loose-coupling and object-oriented concepts like inheritance, polymorphism, encapsulation, composition, interfaces, abstract and concrete classes, abstract and static methods and so forth.

I use Java examples in this book. I find that Java tends to be easy for most object-oriented programmers to read. I've gone out of my way to not use fancy Java features, so whether you code in C++, C#, VB.NET, Python, Ruby, Smalltalk or some other object-oriented language, you ought be able to understand the Java code in this book.

This book is closely tied to Martin Fowler's classic book, Refactoring [F]. It contains references to low-level refactorings, such as

- Extract Method [F]
- Extract Interface [F]
- Extract Superclass [F]
- Extract Subclass [F]
- Pull Up Method [F]
- Move Method [F]
- Rename Method [F]

It also contains references to more sophisticated refactorings, such as

- Replace Inheritance with Delegation [F]
- Replace Conditional with Polymorphism [F]
- Replace Type Code with Subclasses [F]

To understand the pattern-directed refactorings in this book, you don't need to know every refactoring listed above. Instead, you can follow the example code that illustrates how the above refactorings are implemented. However, if you want to get the most out of this book, I do recommend that you have Refactoring [F] close by your side. It's an invaluable refactoring resource, as well as a useful aid for understanding this book.

The patterns I write about come from the classic book, Design Patterns [DP], as well as from authors such as Kent Beck, Bobby Woolf and myself. These are patterns that my colleagues and I have refactored to, towards or away from on real-world projects. By learning the art of pattern-directed refactorings, you'll understand how to refactor or towards patterns not mentioned in this book.

You don't need expert knowledge of these patterns to read this book, though some knowledge of patterns is useful. To help you understand the patterns I've written about, this book includes brief pattern summaries, UML sketches of patterns and many example implementations of patterns. To get a more detailed understanding of the patterns, I'd recommend that you study this book in conjunction with the patterns literature I reference.

This book uses UML 2.0 diagrams. If you don't know UML very well, you're in good company. I know the basics. Whie writing this book, I kept the third edition of Fowler's UML Distilled close by my side and referred to it often.

Signs you should refactor

- code is **duplicated**
- a routine is **too long**
- a loop is too long or **deeply nested**
- a class has poor **cohesion**
- a class uses too much **coupling**
- inconsistent level of **abstraction**
- too many **parameters**
- to **compartmentalize** changes (change one place → must change others)
- to modify an **inheritance hierarchy** in parallel
- to **group related data** into a class
- a **"middle man"** object doesn't do much
- **poor encapsulation** of data that should be private
- a **weak subclass** doesn't use its inherited functionality
- a class contains **unused code**

Some types of refactoring

- refactoring to fit design **patterns**
- **renaming** (methods, variables)
- **extracting** code into a method or module
- **splitting** one method into several to improve cohesion and readability
- changing method **signatures**
- performance **optimization**
- **moving** statements that semantically belong together near each other
- naming (extracting) "magic" **constants**
- **exchanging idioms** that are risky with safer alternatives
- **clarifying** a statement that has evolved over time or is unclear

REFERENCES

1. www.google.com
2. www.wikipedia.org
3. www.studymafia.org