Refactoring

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

What is Refactoring?

“Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which “too small to be worth doing”. However the cumulative effect of each of these transformations is quite significant. “

– Fowler et al. Refactoring. 2018 [1]

What purpose does it serve?

- Continual improvement of your designs.
- Acknowledges that changes don’t happen in isolation; often the cumulative effects of software changes over time can cause code decay.
- Focuses on the transformations themselves, and how to perform them without compromising your source code.

Refactoring for Clean Code

Martin (2008) would say that refactoring produces a “clean” codebase that can be adapted over time as requirements change.

Goals for our code:

- Clear and easy to read: variable names that make sense, no “magic number”, classes that aren’t bloated, well-constructed methods with no side effects.
- Simple: no unnecessary complexity that makes difficult to understand.
- Robust: resilient to change, and less likely to break when changed.
- Intentionally designed: carefully segmented and structured.
- Well-tested: tests demonstrate that the code is working correctly.

Benefits of Refactorings

Improved structure

- Cleaning up class interfaces and relationships.
- Fixing issues with class cohesion/coupling.
- Reducing or removing unnecessary dependencies.
- Removing code duplication.

Improved readability

- Simplifying code to reduce unnecessary complexity.
- Making code more understandable and readable.

Ensuring correctness

- Adding more exhaustive tests.

Refactoring & Testing

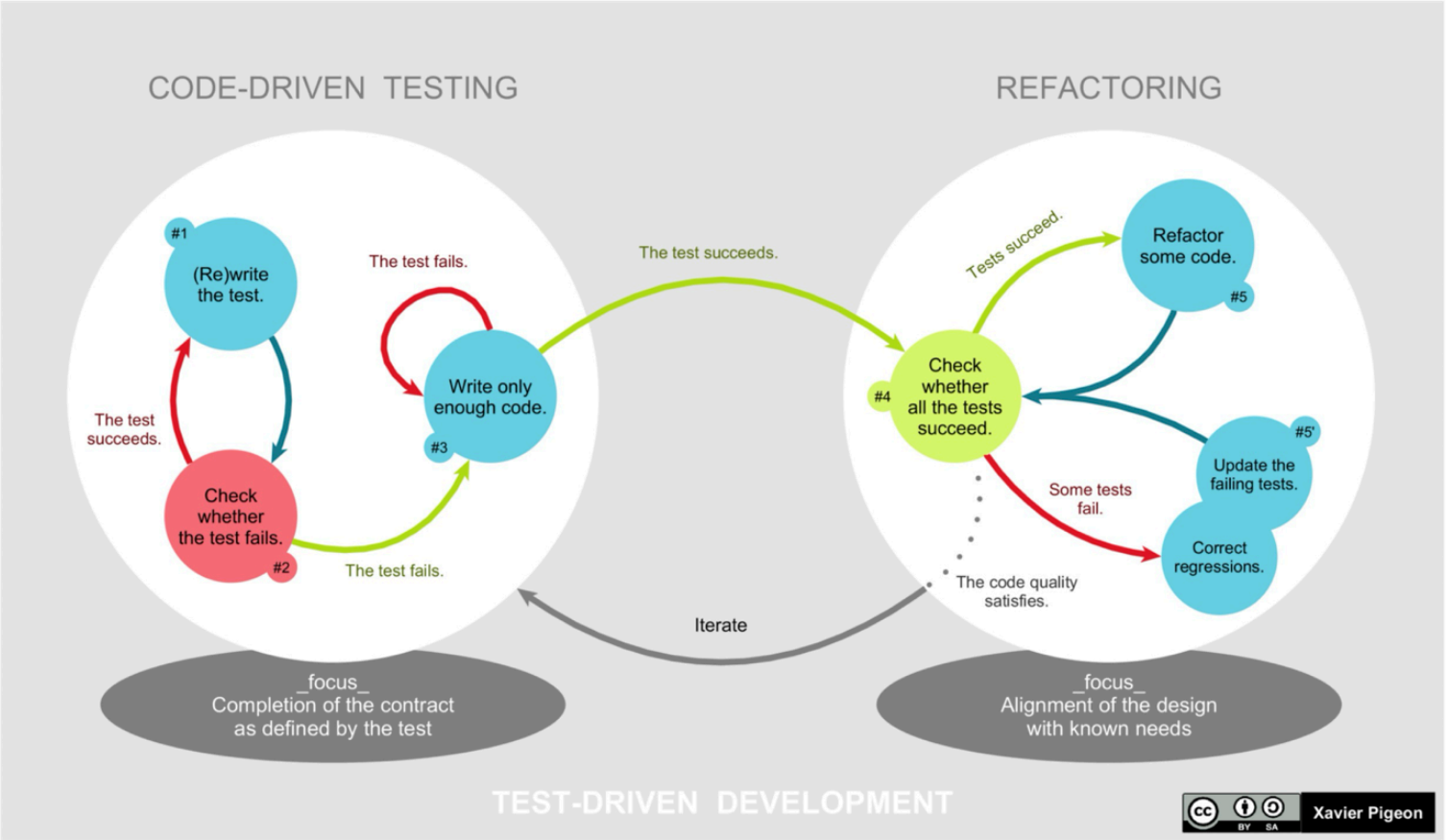


Figure 1: Unit testing should give you confidence that you will not break existing functionality when you refactor your code. Do not refactor unless you have adequate tests in-place!

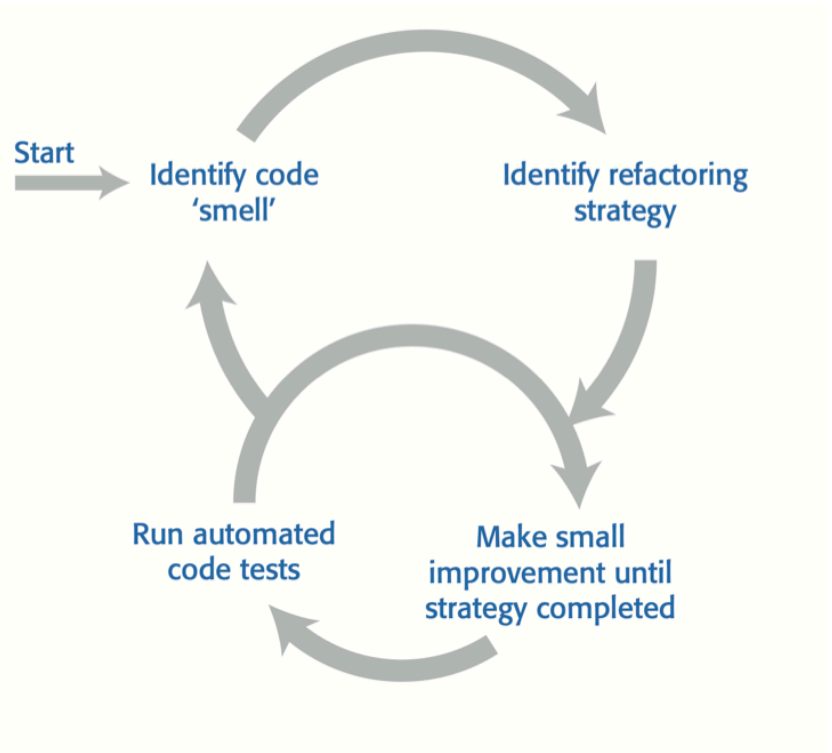
Refactoring Patterns

Code Smells?

Martin Fowler, a refactoring pioneer, suggests that the starting point for refactoring should be to identify code ‘smells’.

Code smells are indicators in the code that there might be a deeper problem.

For example, very large classes may indicate that the class is trying to do too much. This probably means that its structural complexity is high.



Code Smells?

Examples of Poor Structure

Large classes

- This suggests that the single responsibility principle is being violated. Break down large classes into easier-to-understand, smaller classes.

Long methods/functions

- Long methods or functions may indicate that the function is doing more than one thing. Split into smaller, more specific functions or methods.

Duplicated code

- Adds unecessary comlpexity. Rewrite to create a single instance.

Meaningless names

- These make the code harder to understand. Replace with meaningful names and check for other shortcuts that the programmer may have taken.

Unused code

- This increases the reading complexity of the code. Delete it!

Code Smells?

Examples of Excessive Complexity

Reading complexity

- Rename variable, function and class names throughout your program to make their purpose more obvious.

Structural complexity

- Break long classes or functions into shorter units that are likely to be more cohesive than the original large class.

Data complexity

- Simplify data by changing your database schema or reducing its complexity. For example, you can merge related tables in your database to remove duplicated data held in these tables.

Decision complexity

- Replace a series of deeply nested if-then-else statements with guard clauses.

How to Refactor

1. Identify the problem to solve. Apply a pattern to address it.
2. Do not add any new functionality during refactoring.
3. Ensure that all existing tests continue to pass.

There are two case where tests can break down:

1. You made an error during refactoring. This one is a no-brainer: go ahead and fix the error.
2. Your tests were too low-level. For example, you were testing private methods of classes. In this case, the tests are to blame. You can either refactor the tests themselves or write an entirely new set of higher-level tests.

How to Refactor

Figure 2: Martin Fowler maintains a repository of patterns on Refactoring.com [2].

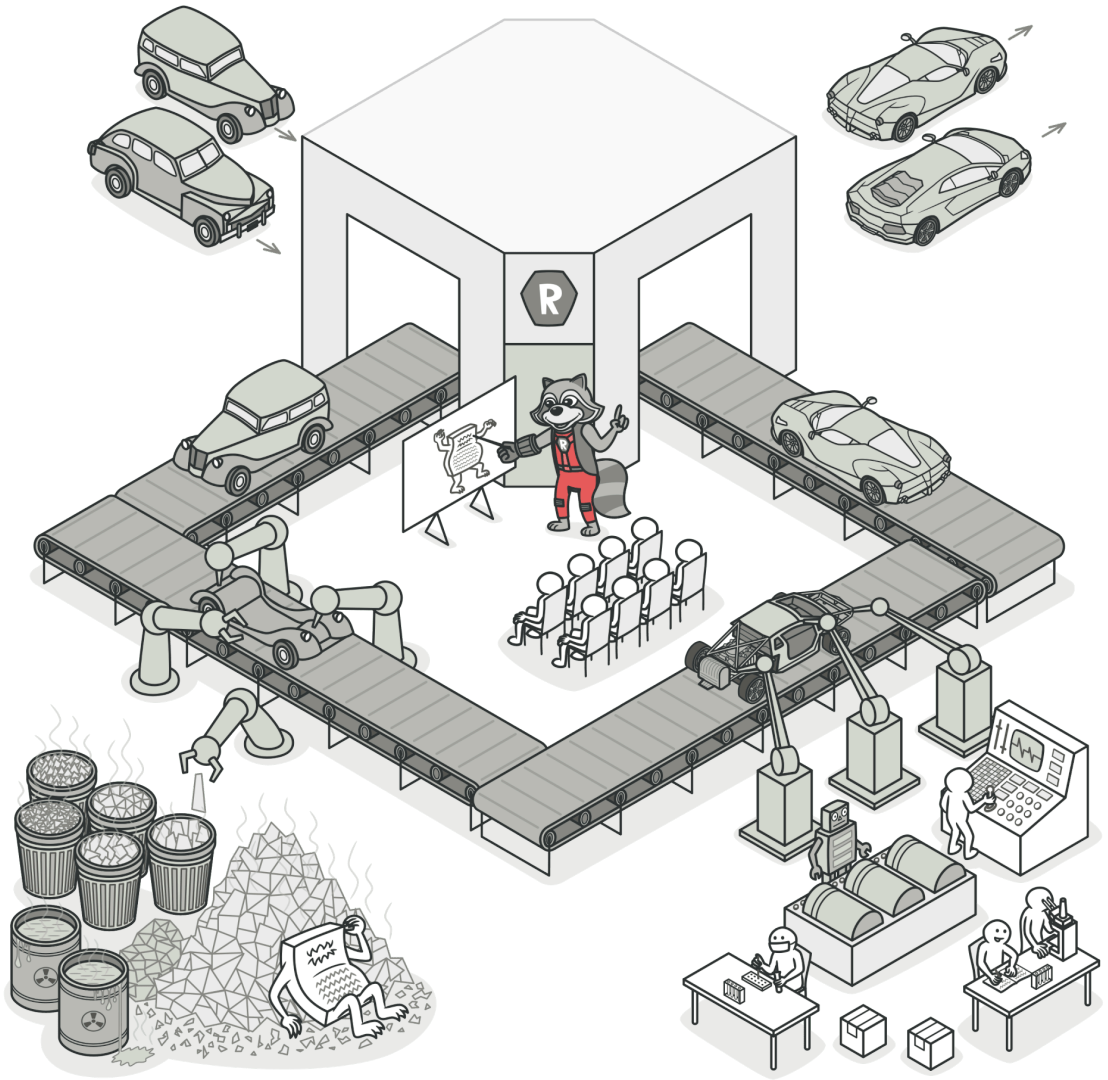


Figure 3: Refactoring.guru also maintains a comprehensive list of techniques, along with detailed examples [3].

Examples

Extract Method

We might extract a method from existing code. Do this to make the original higher-level function easier to read, make it easier to invoke a function.

```
// original
fun printOwing(
  name: String, amount: Double
) {
  // function to print banner
  printBanner()

  // inline print details
  println("name: $name")
  println("amount: $amount")
}
```

```
// refactored
fun printOwing(
  name: String, amount: Double
) {
  printBanner();
  printDetails(name, amount);
}

fun printDetails (
  name: String, amount: Double
) {
  println("name: $name")
  println("amount: $amount")
}
```

Examples

Inline Method

We might also the opposite: remove a pointless method.

Do this when indirection is needless (simple delegation). Also do this when group of methods are badly factored and grouping them makes them clearer.

```
// original
fun getRating(): Int {
  return moreThanFive() ? 2 : 1
}

fun moreThanFive(): Boolean {
  return _numLate > 5
}
```

```
// refactored
fun getRating(): Int {
  return (_numLate > 5) ? 2 : 1;
}
```

Examples

Extract class

You have one class doing work that should be done by two. Create a new class and move the relevant fields and methods from the old class into the new class.

Do this when subsets of methods seem to belong together, or you have data that could be managed as an independent class.

```
// original
class Person {
  name: String,
  officeAreaCode: Int,
  officeNumber: Int,
  getPhoneNumber()
}
```

```
// refactored
class Person {
  name: String,
  phone_number: PhoneNumber
}

class PhoneNumber {
  areaCode,
  number,
  getPhoneNumber()
}
```

IntelliJ

IntelliJ has built-in support for many refactorings, including:

- **Rename** (Shift+F6): Rename a variable, method, class, or other element and updates all references to it throughout the codebase.
- **Extract Method** (Ctrl+Alt+M): Convert a block of code into a new method.
- **Inline** (Ctrl+Alt+N): Replace method calls with the method's code.
- **Change Signature** (Ctrl+F6): Modify the signature of a method, including parameters, return type, and visibility.
- **Move** (F6): Move classes, methods, or variables to a different package.
- **Extract Variable** (Ctrl+Alt+V): Extract an expression into a new variable.
- **Extract Field** (Ctrl+Alt+F): Extract a selected expression into a new field.
- **Introduce Parameter** (Ctrl+Alt+P): Introduce a new parameter to a method/constructor.
- **Safe Delete** (Alt+Delete): Delete a file/element without breaking references.

Bibliography

[1] M. Fowler and K. Beck, *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[2] M. Fowler, “Refactoring Catalog.” [Online]. Available: https://refactoring.com/catalog/

[3] A. Shvets, “Refactoring Guru.” [Online]. Available: https://refactoring.guru/