Refactoring

CS 346 Application Development

Refactoring

"**Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour. Its heart is a series of small behaviour preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring."

— Martin Fowler, Refactoring. 2000, 2018.

We're doing iterative development; code *should* be continually improved as we work with it!

Refactoring

Refactoring is the process of changing a program to reduce its complexity **without changing the external behaviour of that program**.

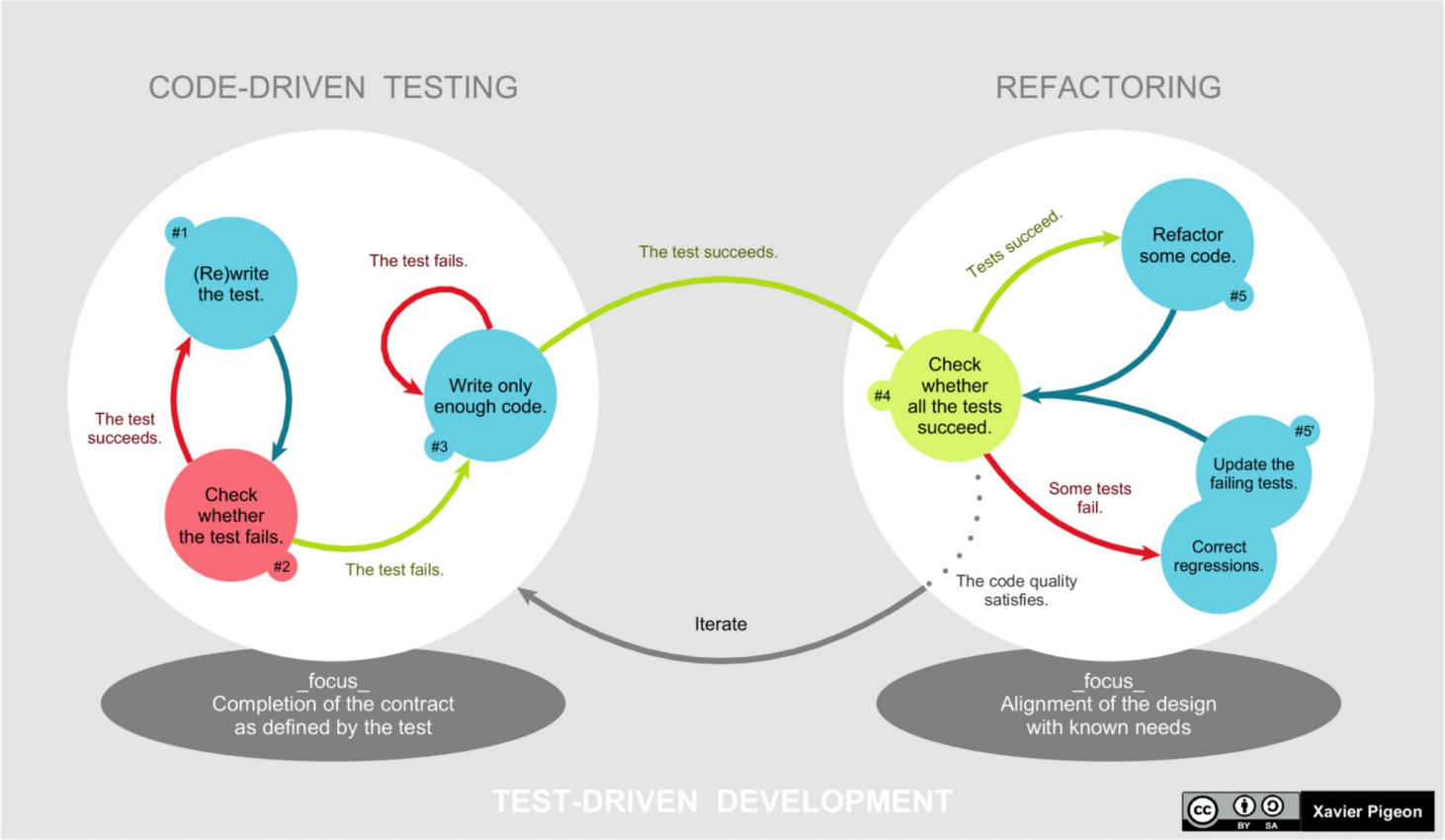
- It makes a program more readable (reducing the ‘reading complexity’) and more understandable.
- It also makes it easier to change, which means that you reduce the chances of making mistakes when you introduce new features.

“Clean Code”

- Martin (2008) would say that refactoring produces a “clean” codebase that can be adapted over time as requirements change.
- A “clean" codebase is:
  - **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 unlikely to break when small changes are introduced.
  - **Intentionally designed**: carefully segmented and structured with no code duplication.
  - **Well-tested**: unit and integration tests demonstrate that the code is working correctly.

What are some examples?

- Cleaning up class interfaces and relationships.
- Fixing issues with class cohesion/coupling.
- Reducing or removing unnecessary dependencies.
- Simplifying code to reduce unnecessary complexity.
- Making code more understandable and readable.
- Adding more exhaustive tests.



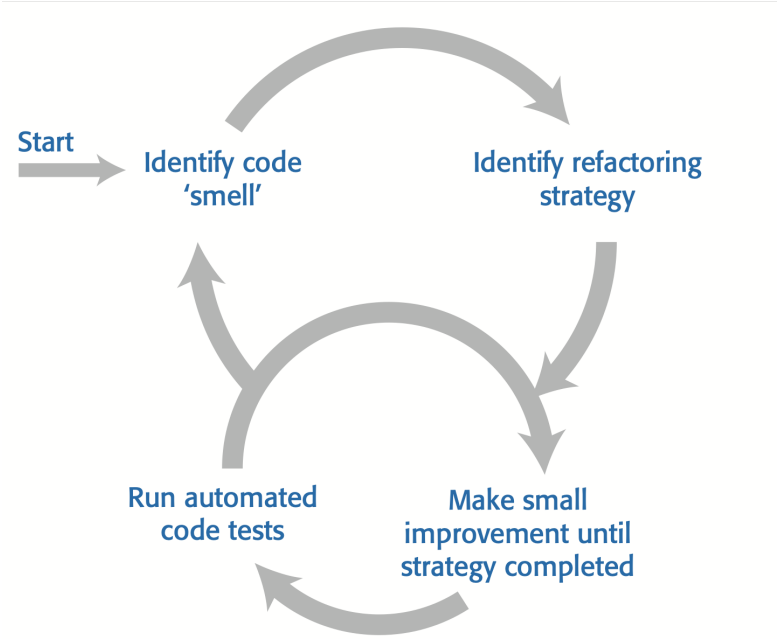
TDD makes refactoring possible. Unit testing should give you confidence that you will not break existing functionality when you refactor.

Code Smells

When to refactor?

Code smells? Time to refactor!

- 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.



Example code smells

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

- Rewrite to create a single instance of the duplicated code that is used as required

Meaningless names

- Theses 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! It's in your Git history right?

Example refactoring for complexity reduction

Reading complexity

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

Structural complexity

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

Data complexity

- You can 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

- You can replace a series of deeply nested if-then-else statements with guard clauses, as I explained earlier in this chapter.

Refactoring Patterns

Goals of refactoring

- **Make the code cleaner**. Use refactoring to rename methods and variables, break apart complex methods into simpler ones, or create new data classes to isolate complexity.
- **Not add any new functionality** during refactoring.
- **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.

Refactoring Patterns

IntelliJ IDEA has built-in support for these operations!

- Martin Fowler. 2018. **Refactoring: Improving the Design of Existing Code**. 2nd Edition. Addison-Wesley. ISBN 978-0134757599. https://refactoring.com/catalog/

6. COMPOSING METHODS
   - 1. Extract Method
   - 2. Inline Method
   - 3. Inline Temp
   - 4. Replace Temp with Query
   - 5. Introduce Explaining Variable
   - 6. Split Temporary Variable
   - 7. Remove Assignments to Parameters
   - 8. Replace Method with Method Object
   - 9. Substitute Algorithm

7. Moving features between elements
   - 10. Move method
   - 11. Move field
   - 12. Extract Class
   - 13. Inline Class
   - 14. Hide Delegate
   - 15. Remove Middle Man
   - 16. Introduce Foreign Method
   - 17. Introduce Local Extension

8. ORGANIZING DATA
   - 18. Self Encapsulate Field
   - 19. Replace Data Value with Object
   - 20. Change Value to Reference
   - 21. Change Reference to Value
   - 22. Replace Array with Object
   - 23. Duplicate Observed Data
   - 24. Change Unidirectional Association to Bidirectional
   - 25. Change Bidirectional Association to Unidirectional
   - 26. Replace Magic Number with Symbolic Constant
   - 27. Encapsulate Field
   - 28. Encapsulate Collection
   - 29. Remove Record with data class
   - 30. Replace Type Code with Class
   - 31. Replace Type Code with Subclasses
   - 32. Replace Type Code with State/Strategy
   - 32. Replace Subclass with Fields

10. MAKING METHOD CALLS SIMPLER
    - 41. Rename method
    - 42. Add Parameter
    - 43. Remove Parameter
    - 44. Separate Query from Modifier
    - 45. Parameterize Method
    - 46. Replace Parameter with Explicit Methods
    - 47. Preserve Whole Object
    - 48. Replace Parameter with Method
    - 49. Introduce Parameter Object
    - 50. Remove Setting Method
    - 51. Hide Method
    - 52. Replace Constructor with Factory Method
    - 53. Encapsulate Downcast
    - 54. Replace Error Code with Exception
    - 55. Replace Exception with Test

https://github.com/HugoMatilla/Refactoring-Summary

Example: extract Method

- We might extract a method from existing code.
- *Do this to make the original higher-level function is easier to read, or to improve the ability of a function to be called from elsewhere in the code.*

```
// original
fun printOwing(name: String, amount: Double) {
  printBanner()
  //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")
}
```

https://github.com/HugoMatilla/Refactoring-Summary - 1-extract-method

Example: 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 moreThanFiveLateDeliveries() ? 2 : 1
}


fun moreThanFiveLateDeliveries(): Boolean {
    return _numberOfLateDeliveries > 5
}
```

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

https://github.com/HugoMatilla/Refactoring-Summary#2-inline-method

Example: Move Method

- A method is using or used by more features of another class than the class on which it is defined. *Do this when classes collaborate too much and are highly coupled.*
- Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation or remove it altogether.

```
// original
class Class1 {
    method()
}

class Class2 {  }
```

```
// refactored
class Class1 {  }

class Class2 {
    method()
}
```

https://github.com/HugoMatilla/Refactoring-Summary#10-move-method

Example: 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,
    officeAreaCode,
    officeNumber,
    getTelephoneNumber()
}
```

```
// refactored
class Person {
    name,
    getTelephoneNumber()
}


class TelephoneNumber {
    areaCode,
    number,
    getTelephoneNumber()
}
```

https://github.com/HugoMatilla/Refactoring-Summary#12-extract-class

Example: Remove Middle Man

- A class is doing too much simple delegation. *Get the client to call the delegate directly.*
- *Do this when the "Middle man" (the server) does “too much”.*

```
// original
class ClientClass {
    val person = Person()
    person.doSomething()
}


class Person {
  fun doSomething() {
    val department = Department()
    department.doSomething()
  }
}
```

```
// refactored
class ClientClass {
    val person = Person()
    val department = Department()
    person.doSomething()
    department.doSomething()
}
```

https://github.com/HugoMatilla/Refactoring-Summary#15-remove-middle-man

Example: Introduce Foreign Method

- A server class you are using needs an additional method, but you can't modify the source code for the original class.

```
// original
val newStart = Date(previousEnd.getYear(),previousEnd.getMonth(),previousEnd.getDate()+1)

// refactored: cannot change date class, so add “foreign method”
fun nextDay(date: Date): Date {
    return Date(date.getYear(),date.getMonth(),date.getDate()+1);
}
val newStart = nextDay(previousEnd)

// refactored: extend Date class, using Kotlin features
fun Date.nextDay(): Date {
    return Date(it.getYear(), it.getMonth(),it.getDate()+1);
}
val newStart = previousEnd.nextDay()
```

https://github.com/HugoMatilla/Refactoring-Summary#16-introduce-foreign-method

IntelliJ supports refactoring! (Ctrl-T)

- **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 or class.
- **Extract Variable (Ctrl+Alt+V):** Extract a selected 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.

References

- JetBrains. 2025. kotlin-test documentation.
- Khorikov. 2020. Unit Testing Principles, Practices, and Patterns. Manning. ISBN ISBN 978-1617296277.