

Refactoring



Motivation

Software development often deals with rethinking decisions.

Ultimately causes reworking portions of the code.

Code is not final, it needs to evolve



Why does it evolve?

- Integrating new features
- Optimizing current functionalities
- Extending tests
- Comment your code?!



Common scenario

Almost every time you change the source code you end up *rewriting/reworking* or *restructuring* it. Is that *refactoring*?

Definition

“*Disciplined* technique for restructuring an existing body of code, altering its internal structure *without changing* its external *behavior*.” — Martin Fowler.

Source: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, Second, 2019.

Why is it needed?

- Improves software design
 - Easier to understand
 - More generalizable
 - Code maintenance
- Preventing bugs

Easier to clean a room if only a few things are out of place



When should we refactor?

Rule of Three.

- First time: Just make it work
- Second: Modify it 🤔
- Third: Refactor it 🧑‍🔧

How do we know its refactoring?

- External behaviour cannot change!
- (Unit-) Test have to be in place!
 - otherwise it is a gamble 🤔
- Happens *naturally*

What to refactor!

Code Smells

- Poor naming
- Duplicated code
- Mixed responsibilities
- ...

Types of refactoring

- Rename (function, variable,...)
- Replace
- Extract
- Inline
- Move



Rename

Variables and functions with a meaningful name make the code easier to understand.

```
1 def hw():
2     print("Hello world!")
3
4
5 t = 3
6 for i in range(t):
7     hw()
```

```
1 def helloworld(): #renamed function
2     print("Hello world!")
3
4
5 num_times = 3 #renamed variable
6 for i in range(num_times):
7     helloworld()
```

Code smell: Mysterious/Poor naming



Replace

Find an equivalent way to replace a block of code

```
1 def hello(something: str | None = None):
2     if something:
3         print(f"Hello {something}!")
4     else:
5         print("Hello!")
6
7
8 hello()
9 hello("world")
10 hello("everyone")
11 hello("class")
12 hello("humans")
13 hello(2025)
```

```
1 def hello(something: str | None = None):
2     if something:
3         print(f"Hello {something}!")
4     else:
5         print("Hello!")
6
7
8 for word in (None, "world", "everyone", "cla
9     hello(word)
```

(Minor) Code duplication: Repeated calls

Extract Refactoring

Divide to conquer

```

1 prompt = """Welcome to the GSS course!
2 Today's lecture is about refactoring.
3 What is your name? """
4 reply = input(prompt)
5 print("Hello", reply.strip())

```

```

Welcome to the GSS course!
Today's lecture is about refactoring.
What is your name? Bob
Hello Bob!

```

- Welcome
- Question
- Greeting

```

1 def welcome():
2     message = """Welcome to the GSS course!
3 Today's lecture is about refactoring."""
4     print(message)
5
6
7 def ask_name():
8     name = input("What is your name? ")
9     return name.strip()
10
11
12 def say_hello(name=""):
13     print(f"Hello {name}!")
14
15
16 welcome()
17 name = ask_name()
18 say_hello(name=name)

```

Code smell: Mixed responsibilities



Inline Refactoring

It's the opposite of extract! If we only do it once, might as well not have it encapsulated in a function.

```
1 def welcome():
2     message = """Welcome to the GSS course!
3 Today's lecture is about refactoring."""
4     print(message)
5
6
7 def ask_name():
8     name = input("What is your name? ")
9     return name.strip()
10
11
12 def say_hello(name=""):
13     print(f"Hello {name}!")
14
15
16 welcome()
17 name = ask_name()
18 say_hello(name=name)
```

```
1 def welcome_ask_name():
2     message = """Welcome to the GSS course!
3 Today's lecture is about refactoring."""
4     print(message)
5     name = input("What is your name? ")
6     return name.strip()
7
8
9 def say_hello(name=""):
10     print(f"Hello {name}!")
11
12
13 name = welcome_ask_name()
14 say_hello(name=name)
```

Code smell: Over decomposition



Moving

Group functionality in a module.
E.g. move `is_prime` to another file

myscript.py

```

1 def is_prime(n: int):
2     for i in range(2, n):
3         if n % i == 0:
4             return False
5     return True
6
7 limit = int(
8     input("""Printing first N primes.
9 Enter N: """))
10 )
11 count, number = (0, 0)
12 while count < limit:
13     if is_prime(number):
14         count += 1
15         print(number)
16     number += 1

```

primes.py

```

1 def is_prime(n: int):
2     for i in range(2, n):
3         if n % i == 0:
4             return False
5     return True

```

myscript.py

```

1 from primes import is_prime
2
3 limit = int(
4     input("""Printing first N primes.
5 Enter N: """))
6 )
7 count, number = (0, 0)
8 while count < limit:
9     if is_prime(number):
10         count += 1
11         print(number)
12     number += 1

```

Potential code smell: Multiple responsibilities.



Refactoring examples

```

1 def fibonacci(n):
2     if n <= 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         previous = 0
8         current = 1
9         for i in range(n - 1):
10            ith = previous + current
11            previous = current
12            current = ith
13         return current

```

The Fibonacci sequence is defined by the recurrence relation:

$$F(n) = F(n - 1) + F(n - 2)$$

with initial numbers:

$$F(0) = 0, \quad F(1) = 1$$

Resulting in: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Refactoring examples

```

1 def fibonacci(n):
2     if n <= 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         previous = 0
8         current = 1
9         for i in range(n - 1):
10            ith = previous + current
11            previous = current
12            current = ith
13     return current

```

```

1 def fibonacci(n):
2     if n <= 0:
3         return 0
4     previous = 0
5     current = 1
6     for i in range(n - 1):
7         ith = previous + current
8         previous = current
9         current = ith
10    return current

```

Prune redundant conditions



Refactoring examples

```
1 def fibonacci(n):
2     if n <= 0:
3         return 0
4     previous = 0
5     current = 1
6     for i in range(n - 1):
7         ith = previous + current
8         previous = current
9         current = ith
10    return current
```

```
1 def fibonacci(n):
2     if n <= 0:
3         return 0
4     previous, current = 0, 1
5     for i in range(n - 1):
6         previous, current = current,
7         return current
```

One-liner update of `previous` and `current` using swap



Refactoring examples

```
1 def fibonacci(n):
2     """
3     Computes the Nth number of the Fibonacci Sequence
4     """
5     if n <= 0:
6         return 0
7     previous, current = 0, 1
8     for i in range(n - 1):
9         previous, current = current, previous + current
10    return current
```

```
1 help(fibonacci)
```

Help on function fibonacci in module fib:

fibonacci(n)

Computes the Nth number of the Fibonacci Sequence

Adding documentation via *docstring* ([PEP-257](#))



Refactoring examples

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     return fibonacci(n - 2) + fibonacci(n - 1)
```

Rewrite the implementation



What was missing?



Development approach

- Write software that solves current needs
 - Only add functionality when you need it
 - *yagni* - you aren't going to need it
- New code? New tests!!!
 - Test-Driven Development highly incentivized



Refactoring on a team

- Each member should be able to refactor independently
- Continuous Integration (CI) shows results by:
 - automatically test code (Unit Tests)
 - pointing where the problems are



Hands-on

- Find code smells in `report.py`.
- Refactor the function `grades_report`.
- Run `test_grades_report.py` (with `pytest`) to ensure behaviour remains.



Hands-on

report.py

```

1 def get_grades():
2     students = list()
3     grades = list()
4     while True:
5         name = input("Insert student name (quit to
6             if "quit" in name:
7                 return students, grades
8             grade = int(input(f"Insert {name}'s grade:"))
9             students.append(name)
10            grades.append(grade)
11
12
13 def grades_report(students, grades):
14     average = 0
15     maxgrade = 0
16     mingrade = 100
17     counter = 0 # do we need this variable?
18     for grade in grades: # can this be improved?
19         counter += 1 # len(grades) will give the n
20
21     # do we need this? we can just return avg, max
22     report = {"students": students, "grades": grade
23     for i in range(counter):
24         grade = grades[i]
25

```

test_grades_report.py

```

1 from report import grades_report, format_report
2
3
4 def test_empty_report():
5     students = list()
6     grades = list()
7     assert (None, None, None) == grades_report(stud
8
9
10 def test_grades_report():
11     students = ["Alice", "Bob"]
12     grades = [1, 2]
13     assert len(students) == len(grades)
14     size = len(students)
15     assert (
16         1.5,
17         2,
18         1,
19     ) == grades_report(students, grades)
20
21
22 def test_format_report():
23     students = ["Alice", "Bob"]
24     grades = [1, 2]
25

```



Hands-on

```
$ python3 report.py
```

```
Insert student name (quit to finish):Alice
Insert Alice's grade:2
Insert student name (quit to finish):Bob
Insert Bob's grade:1
Insert student name (quit to finish):quit
Report of 2 students:
Alice -> 2
Bob -> 1
  Average: 1.5
  Maximum: 2
  Minimum: 1
```

```
pytest test_grades_report.py
```

```
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5
plugins: anyio-4.4.0
collected 3 items
test_grades_report.py::test_empty_report PASSED [ 33%]
test_grades_report.py::test_grades_report PASSED [ 66%]
test_grades_report.py::test_format_report PASSED [100%]

===== 3 passed in 0.01s =====
```



Legacy Code

- **Dramatic:** code becomes legacy code as soon as it's written
- **General:** code inherited from someone else
- **Complement:** code without tests (specially dangerous!!!)



How to deal with legacy code

- Read it
- Create tests (extensively)
 - Characterization tests are crucial!
- Refactor gradually
- Rewrite new interface (if immutable)
- Deprecate or remove dead code



Take home messages

- You can only safely refactor tested codebases
 - Write tests before touching legacy ones
- Test-driven development & *yagni* are *refactor-friendly*
- Refactor small bits, often



Further Reading

- Fowler, M., & Beck, K. (2019). [Refactoring: Improving the design of existing code \(Second edition.\)](#). Boston: Addison-Wesley.
- [Refactoring guru](#)
 - Code smells
 - Design patterns
 - Refactoring techniques

