COMP642

Object Oriented Programming

Lectorial 11 - Code Refactoring

Why Do Good Developer Write Bad Software?

- Requirements change over time, making it hard to update your code and leading to less optimal designs.
- Time and money cause you to take shortcuts in your coding practices.
- You learn a better way to do something.

What is Refactoring?

- Refactoring (noun)

  A change to the internal structure of software to make it easier to understand and cheaper to modify, without changing its observable behaviour.

- Refactoring (verb)

  The process of restructuring software through a series of internal improvements, without changing its observable behaviour.

- Refactoring modifies software to improve its readability, maintainability, and extensibility without changing what the software does.
- Internal structure is improved.
- External behaviour stays the same.

Why Refactor?

- Improves software design
- Easier to understand
- Helps identify bugs
- Speeds up programming
- Reduces code size
- Boosts maintainability

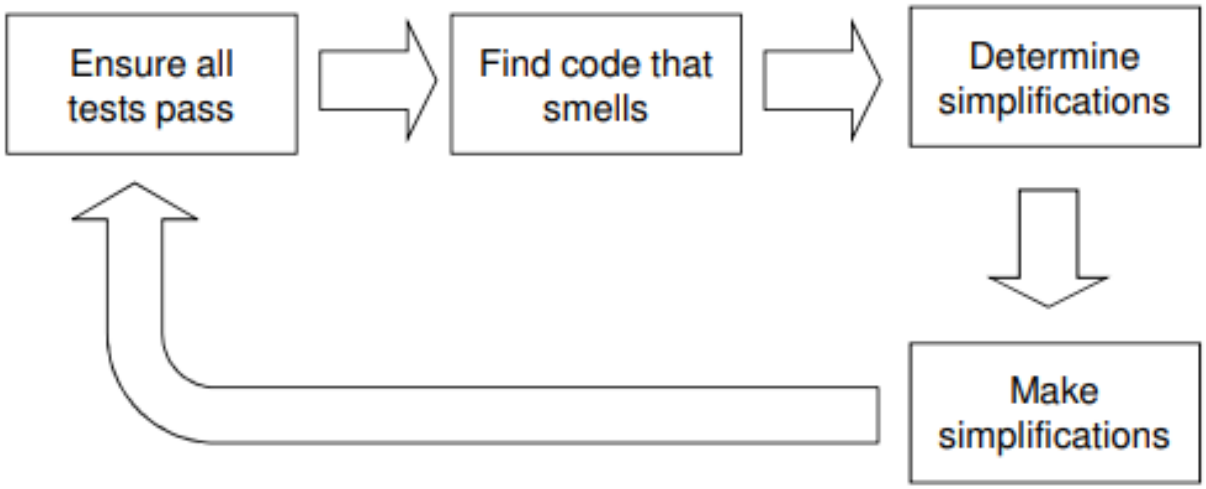
When to Refactor?

- The Rule of Three
- When adding new functions
- When fixing bugs
- During code reviews

When Not to Refactor?

- When tests are failing
- When it is better to rewrite the code
- When facing tight deadlines

How to Refactor?



1. Ensure all tests are passing
2. Identify areas of code that exhibit “smells”
3. Plan ways to simplify the identified code
4. Implement the simplifications
5. Run tests again to verify functionality
6. Continue the simplify/test cycle until all code smells are eliminated

A code smell is a hint that something might be wrong with the code

Code Smells (1)

- Duplicated code - The same code structure appears in more than one place
- Long method - Methods that are too long are hard to understand
- Large class - A class that does too many things
- Long parameter list - Having many parameters makes it confusing to use
- Divergent change - One class is changed in different ways for different reasons
- Shotgun surgery - The opposite of divergent change, where a single change requires multiple small updates across different classes

Code Smells (2)

- Feature envy - A method that seems more interested in a different class than the one it is in
- Data clumps - Groups of data that are frequently used together
- Primitive obsession - Use of primitives instead of small objects for simple tasks
- Complex Switch statements
- Parallel inheritance hierarchies - every time you make a subclass of one class, you also must make a subclass of another
- Lazy class - A class that doesn’t have enough functionality or responsibilities to justify its existence

Code Smells (3)

- Speculative generality - Code to handle things in the future
- Temporary field - Instance variable is set only in certain circumstances
- Message chains - Asking one object to obtain another object
- Middleman - Acts as a delegate, forwarding requests to another class.
- Inappropriate intimacy - One class uses the internal fields of another class
- Alternative classes with different interfaces - Methods that provide similar functionalities but use different method signatures

Code Smells (4)

- Incomplete library class - lacks necessary methods or features that would make it fully functional
- Data class - classes that have fields, getter and setter methods for the fields, and nothing else
- Refused bequest – subclasses don’t want or need what they are given
- Comments - the comments are there because the code is bad

Refactoring Techniques - Extract Method

```
def printOwing(self):
    self.printBanner()

    # print details
    print("name:", self.name)
    print("amount:", self.getOutstanding())
```

Problem:

You have a code fragment that can be grouped together.

Solution:

Move this code to separate new method and replace the old code with a call to the method.

```
def printOwing(self):
    self.printBanner()
    self.printDetails(self.getOutstanding())

def printDetails(self, outstanding):
    print("name:", self.name)
    print("amount:", outstanding)
```

Refactoring Techniques - Replace Temp with Query

```
def calculateTotal():
    basePrice = quantity * itemPrice
    if basePrice > 1000:
        return basePrice * 0.95
    else:
        return basePrice * 0.98
```

Problem:

You place the result of an expression in a local variable for later use in your code.

Solution:

Move the entire expression to a separate method and return the result from it.

```
def calculateTotal():
    if basePrice() > 1000:
        return basePrice() * 0.95
    else:
        return basePrice() * 0.98

def basePrice():
    return quantity * itemPrice
```

Refactoring Techniques - Substitute Algorithm

```
def foundPerson(people):
    for i in range(len(people)):
        if people[i] == "Don":
            return "Don"
        if people[i] == "John":
            return "John"
        if people[i] == "Kent":
            return "Kent"
    return ""
```

Problem:

If you want to replace an existing algorithm with a new one?

Solution:

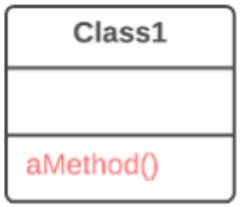
Replace the body of the method that implements the algorithm with a new algorithm.

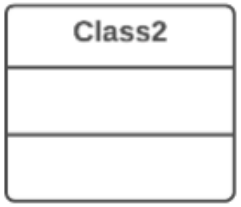
```
def foundPerson(people):
    candidates = ["Don", "John", "Kent"]
    for i in range(len(people)):
        if people[i] in candidates:
            return people[i]
    return ""
```

Refactoring Techniques - Move Method

Problem:

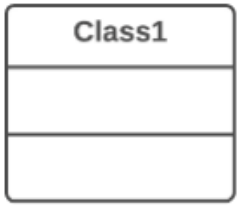
A method is used more in another class than in its own class.

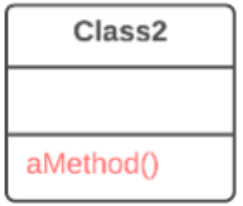




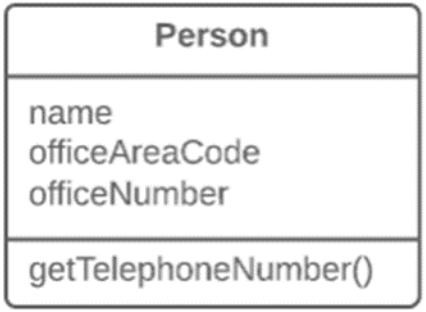
Solution:

Create a new method in the class that uses the method the most, then move code from the old method to there.





Refactoring Techniques - Extract Class

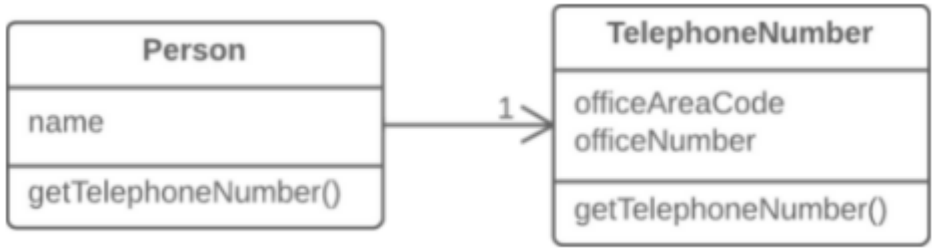


Problem:

When one class does the work of two, awkwardness results.

Solution:

Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.



Refactoring Techniques - Decompose Conditional

```
if date.before(SUMMER_START) or date.after(SUMMER_END):
    charge = quantity * winterRate + winterServiceCharge
else:
    charge = quantity * summerRate
```

Problem:

You have a complex conditional (if-then/else or switch)

Solution:

Decompose the complicated parts of the conditional into separate methods: the condition, then and else.

```
if isSummer(date):
    charge = summerCharge(quantity)
else:
    charge = winterCharge(quantity)
```

Refactoring Techniques - Consolidate Duplicate Conditional Fragments

```
if isSpecialDeal():
    total = price * 0.95
    send()
else:
    total = price * 0.98
    send()
```

Problem:

Identical code can be found in all branches of a conditional.

Solution:

Move the code outside of the conditional.

```
if isSpecialDeal():
    total = price * 0.95
else:
    total = price * 0.98
send()
```

Refactoring Techniques - Preserve Whole Object

```
low = daysTempRange.getLow()
high = daysTempRange.getHigh()
withinPlan = plan.withinRange(low, high)
```

Problem:

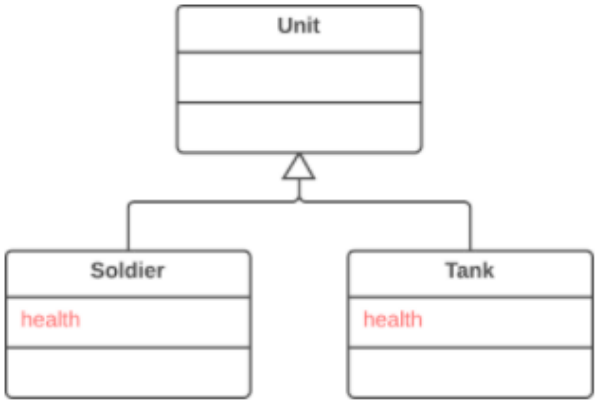
You get several values from an object and then pass them as parameters to a method.

Solution:

Instead, try passing the whole object.

```
withinPlan = plan.withinRange(daysTempRange)
```

Refactoring Techniques - Pull Up Field

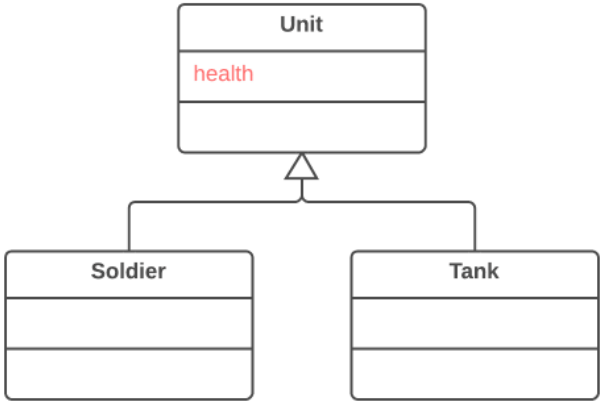


Problem:

Two classes have the same field.

Solution:

Remove the field from subclasses and move it to the superclass.



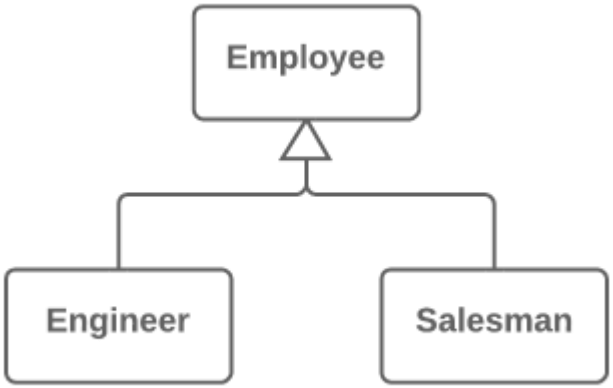
Refactoring Techniques - Replace Type Code with Subclasses

Problem:

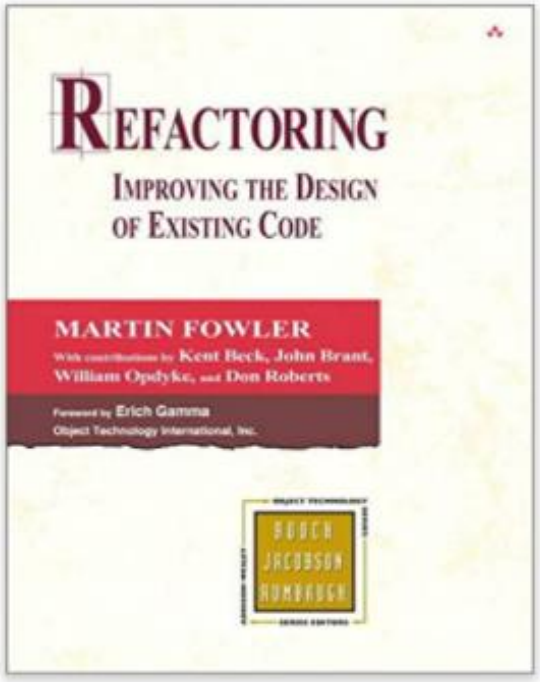
You have a coded type that directly affects program behaviour (values of this field trigger various code in conditionals).

Solution:

Create subclasses for each value of the coded type. Then extract the relevant behaviours from the original class to these subclasses.



Further Reading on Refactoring Techniques



https://refactoring.guru/refactoring

Adding Safety

- Refactoring may introduce new bugs into the system.
- To minimise this risk, use automated tests regularly throughout the process.
- After each refactor, run your test suite to verify that the system’s functionality remains unchanged.