Engenharia de Software
(14341, 16230, 15386)

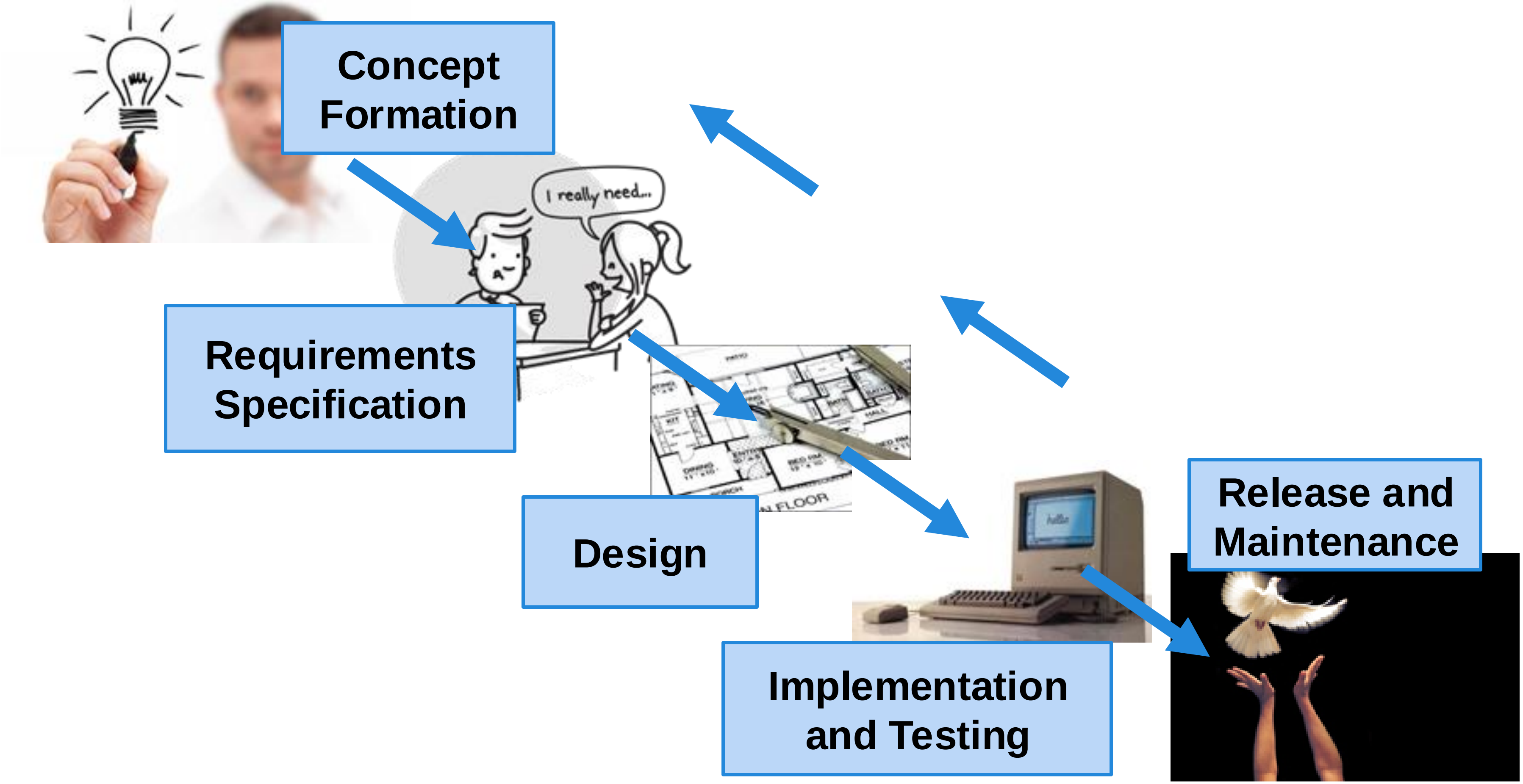
Refactoring

(adapted from lecture notes of the “DIT 635 - Software Quality and Testing” unit,
delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

Today’s Goals

- Cover the basics of refactoring
- Introduce the idea of “code smells”

The Software Lifecycle



Concept Formation

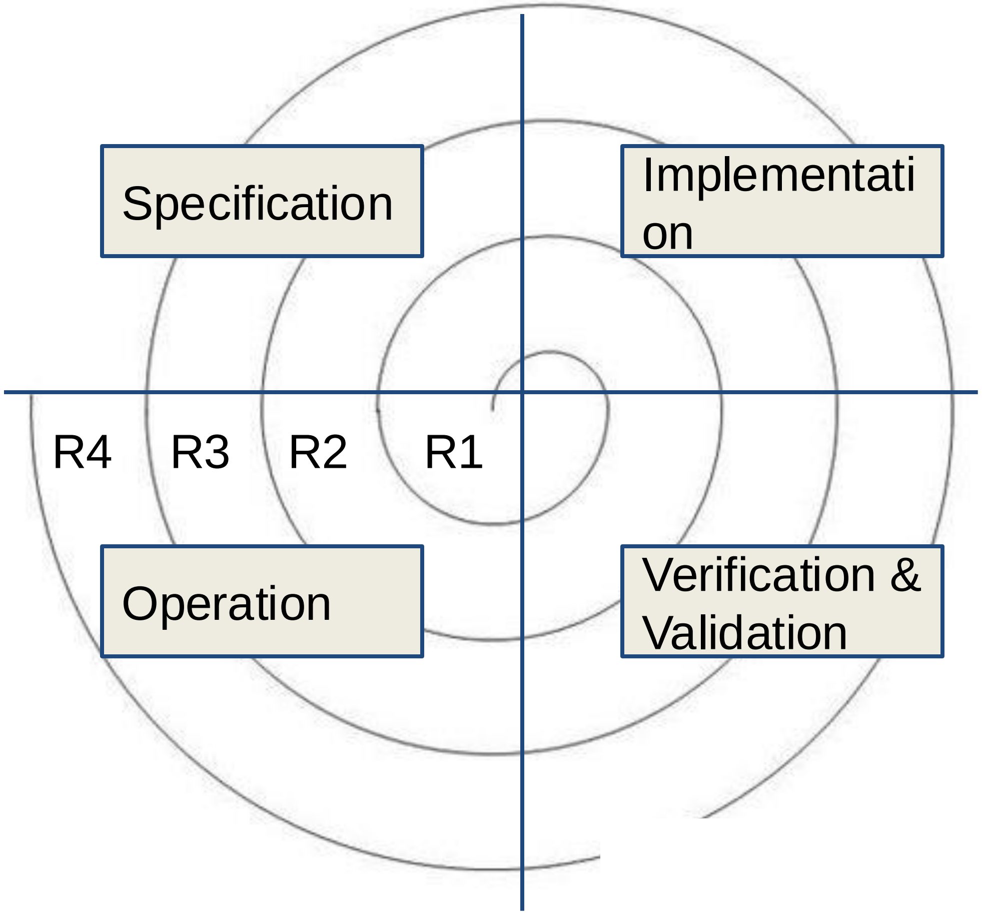
Requirements Specification

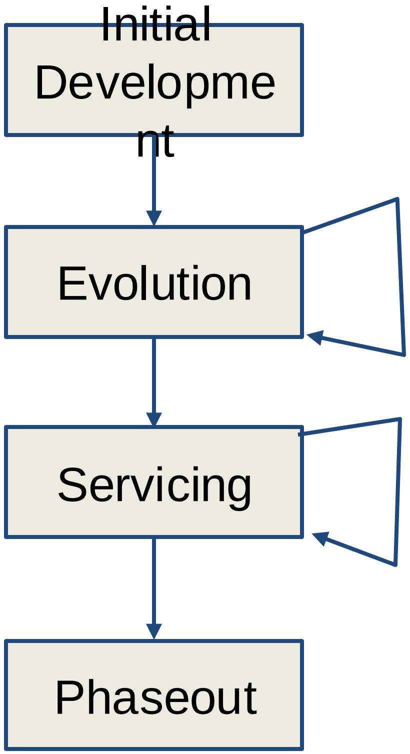
Design

Implementation and Testing

Release and Maintenance

The Real Lifecycle





Software Maintenance

- Fault Repairs
  - Changes made in order to correct coding, design, or requirements errors.
- Environmental Adaptations
  - Changes made to accommodate changes to the hardware, OS platform, or external systems.
- Functionality Addition
  - New features are added to the system to meet new user requirements.

Software Maintenance Effort

- Maintenance costs more than the initial development.
  - 2/3rds of budget goes to maintenance on average.
  - Up to four times the development cost to maintain critical systems.
- General breakdown:
  - 65% of effort goes to functionality addition
  - 18% to environmental adaptation
  - 17% to fault repair

Maintenance is Hard

It is harder to maintain than to write new code.

- Must understand code written by another developer, or code that you wrote long ago.
- Creates a “house of cards” effect.
- Developers tend to prioritize new development.

Smooth maintenance requires planning and design that supports maintainability.

The Laws of Software Evolution

- Maintenance is an inevitable process.
  - Requirements change as the environment changes.
  - Changing the software causes environmental changes, which leads to more requirement changes.
- As changes occur, the structure degrades.
  - When changes are made, the structure becomes more complex.
  - To prevent this, resources must go into *preventative maintenance* - refactoring to preserve and simplify the structure without adding to functionality.

The Laws of Software Evolution

- The amount of change in each release is approximately constant.
  - The more functionality introduced, the more faults.
  - A large functionality patch tends to be followed by a patch that fixes faults without adding additional functionality. Small functionality changes do not require a fault-correcting patch.
- Functionality must continually increase to maintain user satisfaction.

The Laws of Software Evolution

- The quality of the system will decline unless updated to work with changing environment.
- To improve quality, evolution must be treated as a feedback system.
  - Stakeholders must be continually involved in evolution, and changes should be influenced by their needs.

Refactoring

- Process of revising the code or design to improve its structure, reduce complexity, or otherwise accommodate change.
- When refactoring, you do not add functionality.
- Continuous process of improvement throughout the evolution of the system.

Why Refactor?

Why fix what isn’t broken?

- Components have three purposes:
  - To perform a service.
  - To allow change.
  - To be understood by developers reading it.
- If it does not do any of these, it is “broken”.
- Enables change and improves understandability.

Refactoring is an Iterative Process

- Refactoring should take place as an iterative cycle of small transformations.
  - Choose a small part of the system, redesign it, and make sure it still works.
  - Choose a new section of the system and refactor it.
- Refactoring requires unit tests.
  - Make sure the code works before and after.

Choosing What to Refactor

- Refactor any piece of the system that:
  - Seems to work,
  - But isn’t well designed,
  - And now needs new functionality.
- There are stereotypical situations that indicate the need for refactoring.
  - These are called **“bad smells”**.

Code Smells

- **Code is duplicated** in multiple places.
- A method is **too long**.
- Conditional statements control behavior based on an **object type**.
- Groups of data **attributes are duplicated**.
- A class has **poor cohesion** or **high coupling**.
- A method has **too many parameters**.
- **Speculative generality** - adding functionality that “we might need someday.”

More Code Smells

- Changes must be made in **several places**.
- **Poor encapsulation** of data that should be private.
- If a **weak subclass** does not use inherited functionality.
- If a class contains **unused code**.
- If a class contains **potentially unused attributes** that are only set in particular circumstances.
- There are data classes containing only attributes, getters, and setters, but nothing else - **objects should encapsulate data and behaviors.**
  - Unless that data is used by multiple classes.

Common Refactorings
(more at http://www.refactoring.com)

Composing Methods

- Extract Method
- Inline Method; Inline Temp
- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameters
- Substitute Algorithm

Moving Features Between Objects

- Move Method; Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middleman
- Introduce Foreign Method

Organizing Data

- Replace Data Value with Object
- Change Value to Reference; Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to to Unidirectional

Simplifying Conditional Expressions

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

Making Method Calls Simpler

- Rename Method
- Add/Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

Dealing with Generalization

- Pull Up Field; Method; Constructor Body
- Push Down Method; Push Down Field
- Extract Subclass; Extract Superclass; Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation (or vice versa)

Big Refactorings

- Nature of the Game
- Tease Apart Inheritance
- Convert Procedural Design to Objects
- Separate Domain from Presentation
- Extract Hierarchy

Refactorings - Composing Methods

- If you have a complex code fragment that can exist independently, **extract it into its own method**.
- If you have a method that is extremely simple, **inline** it into locations where it is used.
- If you assign values to a temporary variable more than once, **split it into additional temporary variables**.
- If assignments are made to parameter variables in a method, instead **assign to a temporary variable**.
- If an algorithm is hard to understand, **swap it for a version that is clearer**.

Refactorings - Moving Features Between Objects

- If a method or field is used more by a calling class than the class it is placed in, **move** it.
- If a class is doing more work than it should (or has low cohesion), **extract** a subset of related methods into a new class.
- If a class is doing too little, **combine** it with another.
- If a class delegates too many calls to a middleman class, **get rid of the middleman** and call the client directly.
- If an imported class needs an additional method, but you can’t modify it directly, **create a method in the client class with the imported object as a parameter**.

Refactorings - Conditional Expressions & Data

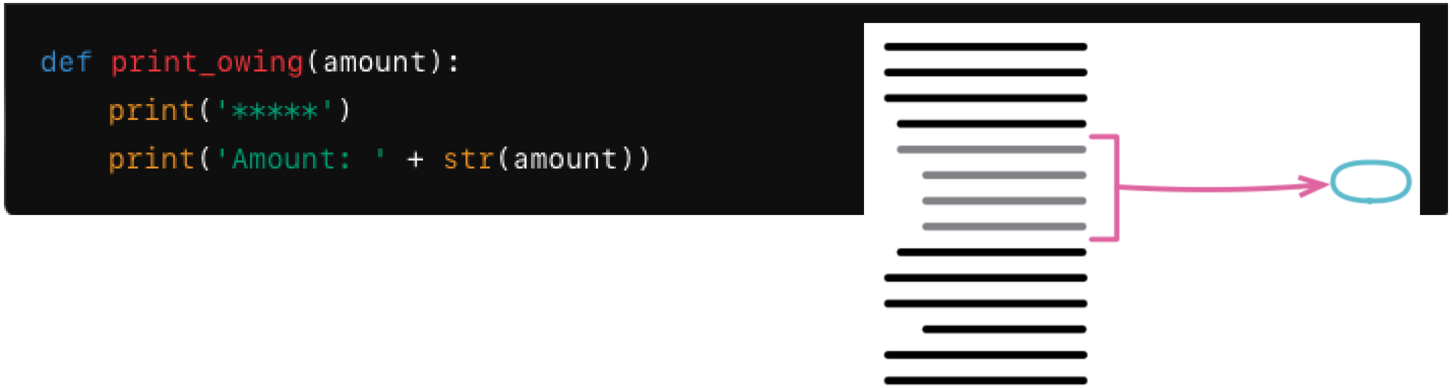
- If your conditional statements are too complex, **extract methods from the if, then, and else conditions**.
- If you have a sequence of conditional tests with the same result or repeated conditions in each branch, **consolidate them into fewer conditional statements**.
- If you have conditional statements to choose behavior based on object type, instead **use polymorphism**.
- If you have an attribute that needs additional data or operations, turn it into a new type of **data object**.
- If certain array values have special meaning, use **a class to store items instead**.

Refactorings - Simplifying Method Calls and Generalization

- If a method both returns a value and changes the state of a passed object, split into two methods and **separate the query from the modifier**.
- If several methods do similar things - differentiated by value - create one method that takes the **value as a parameter**.
- If two classes have the same attribute/method/constructor body, **pull it up into the parent**. If an item is only used by some subclasses, **push it into the children**.
- If a class has features only used situationally, **extract subclasses** for those situations.

Example 1: Extract Method

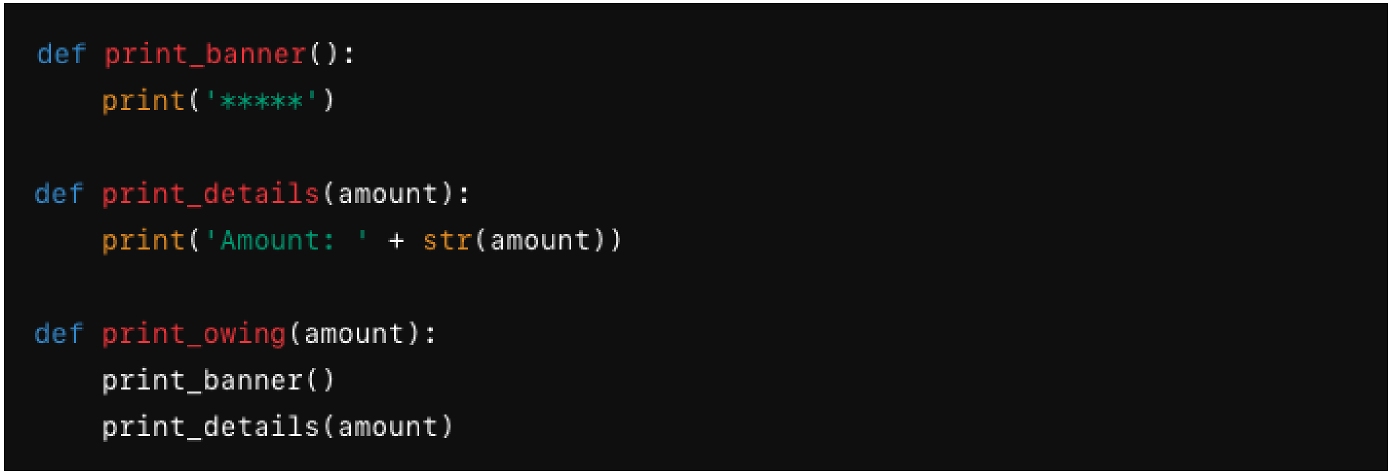
✧ Before: Long method doing multiple things.



```
def print_owing(amount):
    print('*****')
    print('Amount: ' + str(amount))
```

Example 1: Extract Method

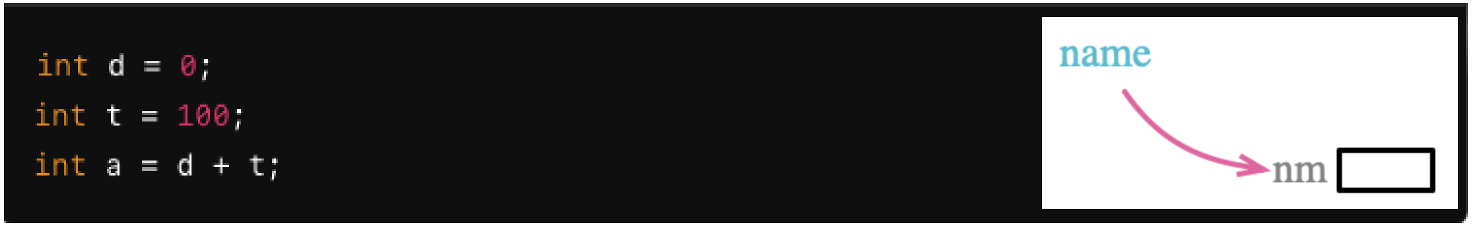
- After: Break the method into smaller, well-named methods.
- Explanation: Improves readability and reusability.



```python
def print_banner():
    print('*****')

def print_details(amount):
    print('Amount: ' + str(amount))

def print_owing(amount):
    print_banner()
    print_details(amount)
```

Example 2: Rename Variable

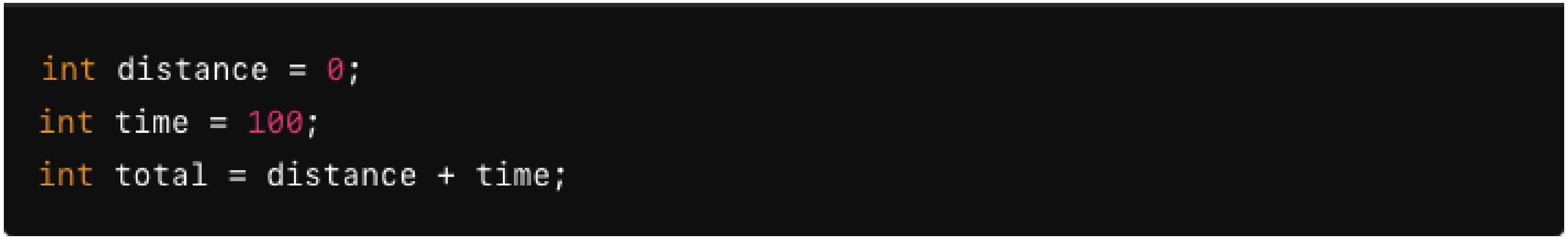
- Before: Unclear or misleading variable names.



```
int d = 0;
int t = 100;
int a = d + t;
```

Example 2: Rename Variable

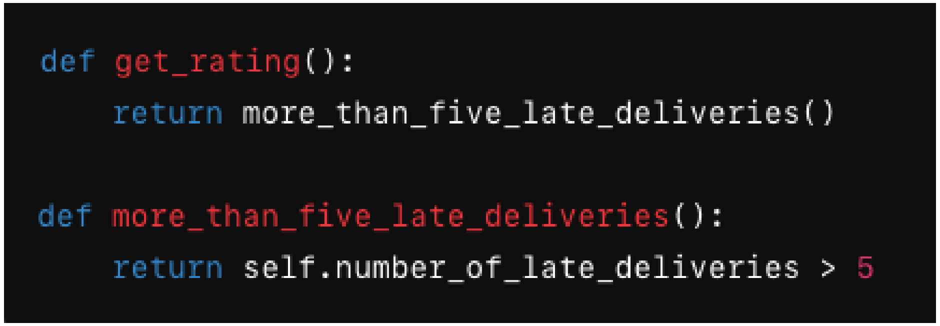
- After: Replace with meaningful, descriptive names.
- Explanation: Enhances code readability and maintainability.



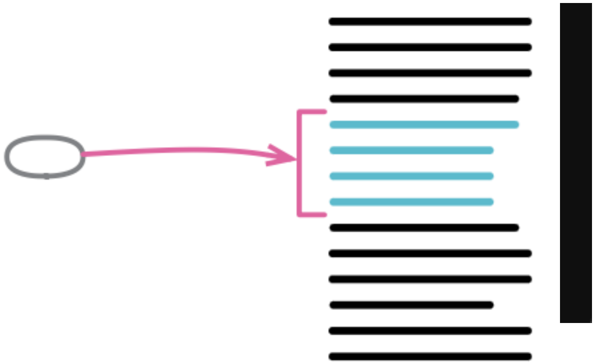
```
int distance = 0;
int time = 100;
int total = distance + time;
```

Example 3: Inline Method

✧ Before: Method that is too simple and used only once.



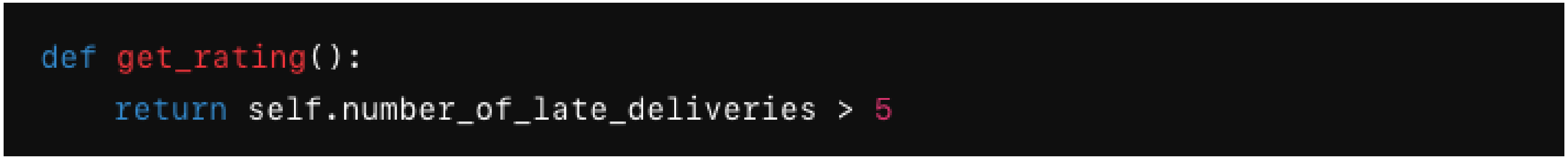
```
def get_rating():
    return more_than_five_late_deliveries()

def more_than_five_late_deliveries():
    return self.number_of_late_deliveries > 5
```



Example 3: Inline Method

✧ After: Inline the method directly into the caller.

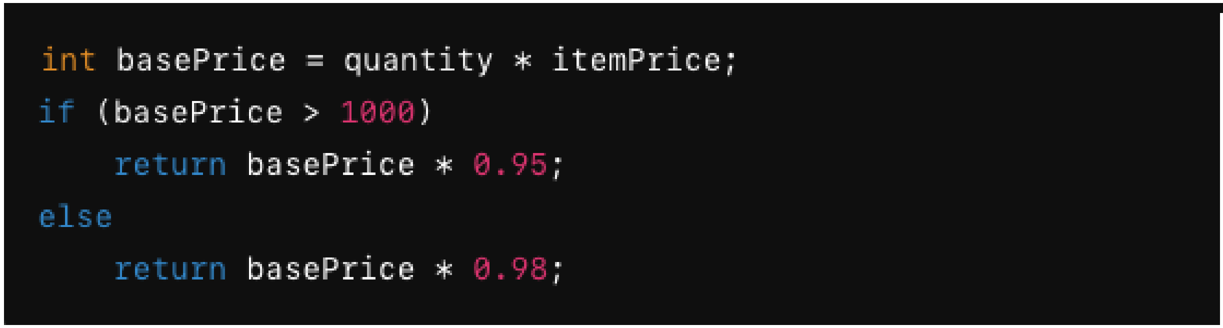
✧ Explanation: Reduces unnecessary indirection.



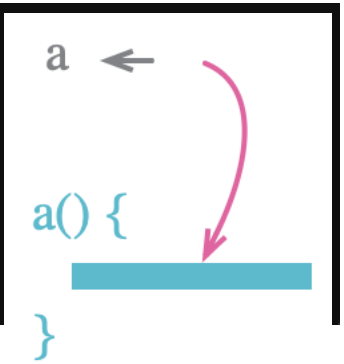
```
def get_rating():
    return self.number_of_late_deliveries > 5
```

Example 4: Replace Temp with Query

✧ Before: Temporary variable holds result of expression.

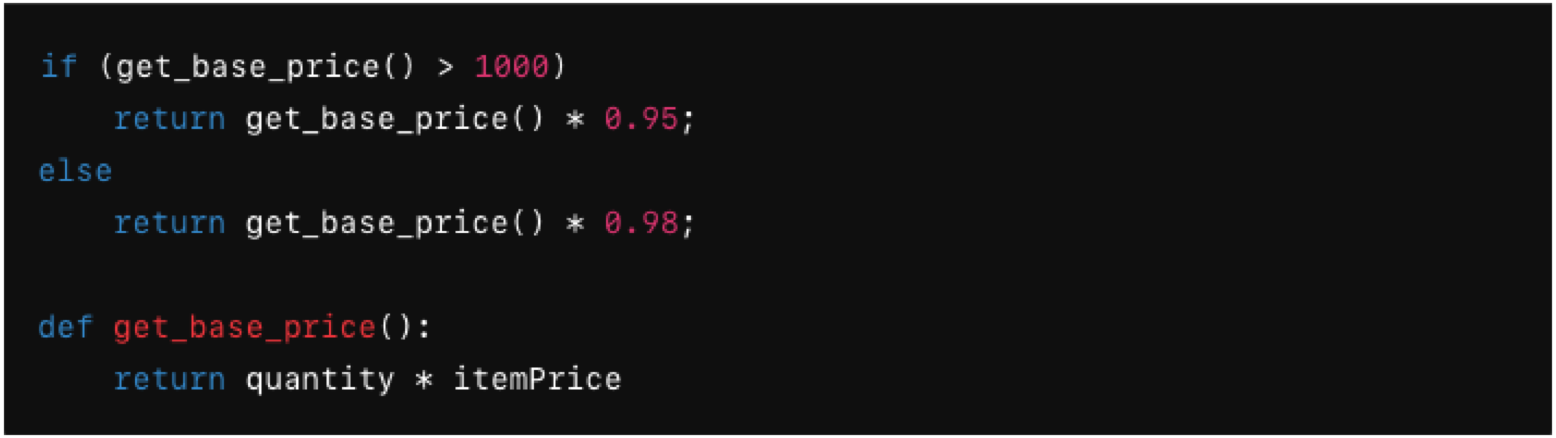


```
int basePrice = quantity * itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```



Example 4: Replace Temp with Query

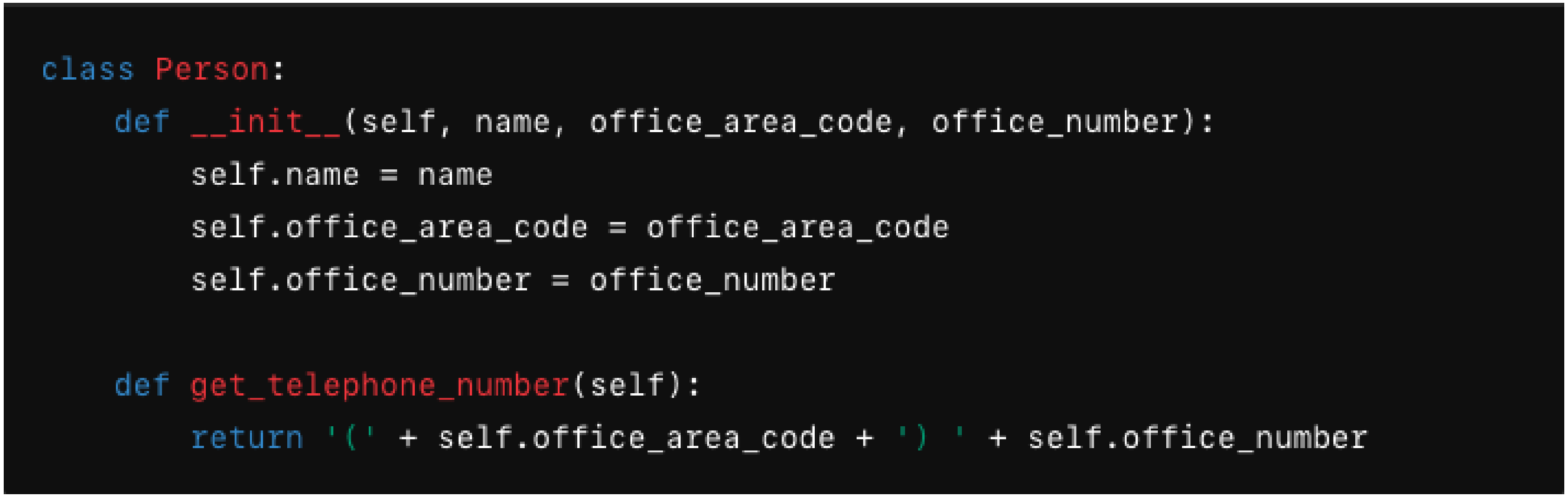
- After: Replace temp with method that directly returns the result.
- Explanation: Makes the code cleaner and easier to understand.



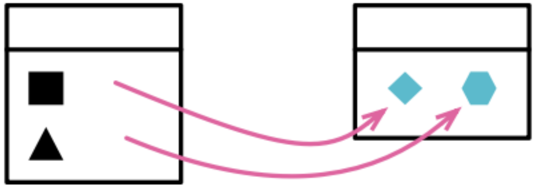
```
if (get_base_price() > 1000)
    return get_base_price() * 0.95;
else
    return get_base_price() * 0.98;

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

Example 5: Extract Class

✧ Before: Class doing too much (God class).

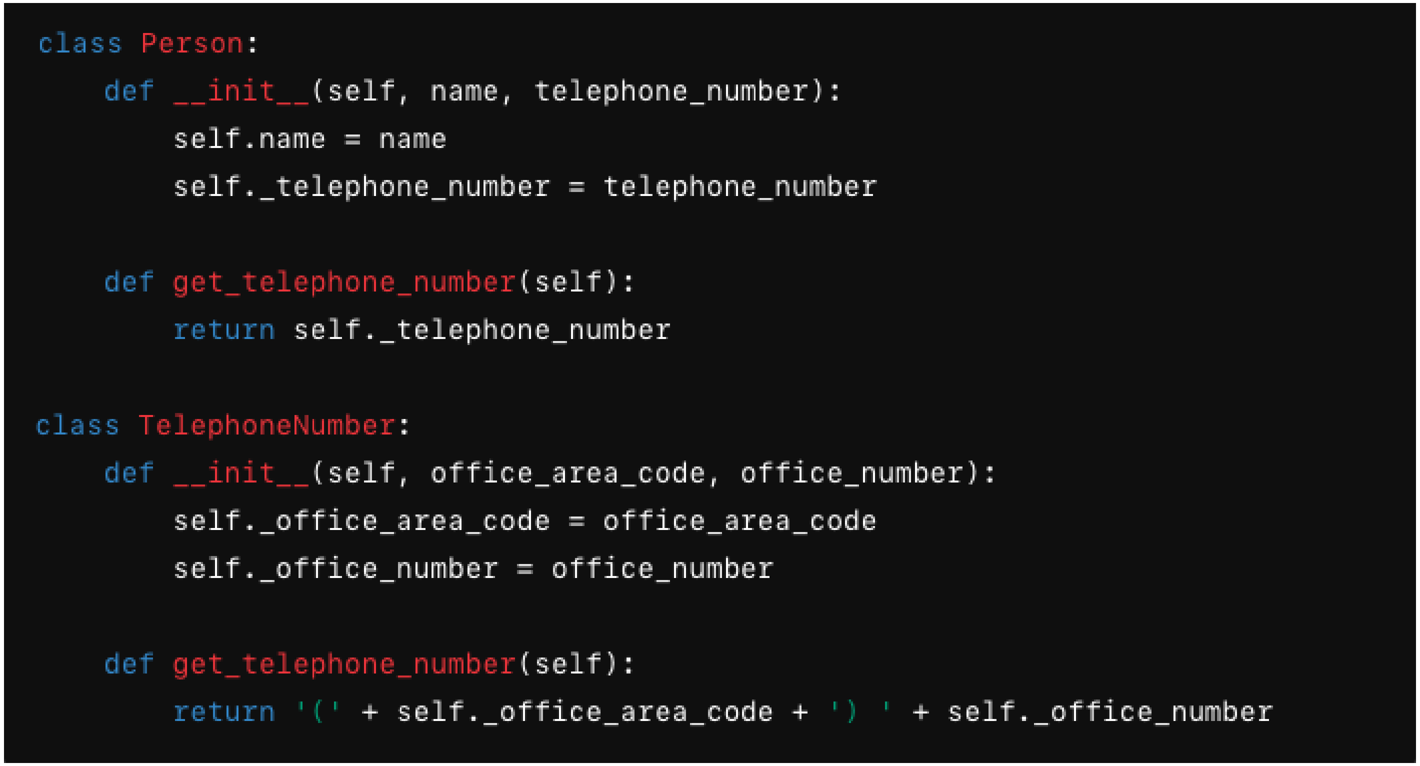


```python
class Person:
    def __init__(self, name, office_area_code, office_number):
        self.name = name
        self.office_area_code = office_area_code
        self.office_number = office_number

    def get_telephone_number(self):
        return '(' + self.office_area_code + ') ' + self.office_number
```



Example 5: Extract Class

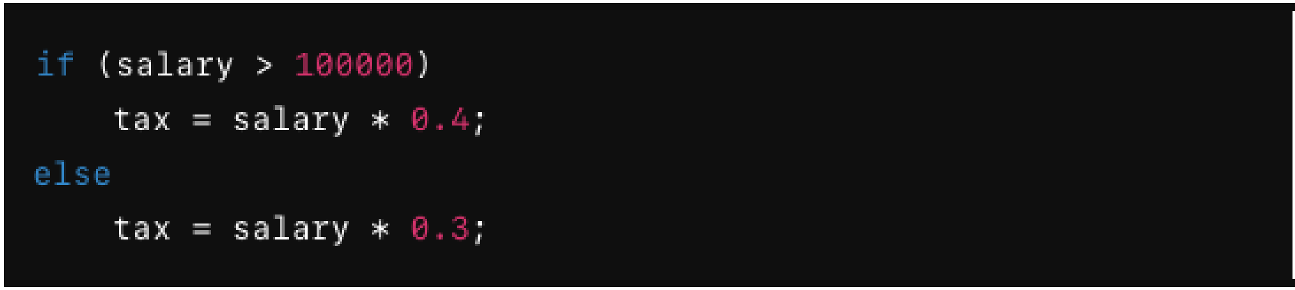
- After: Split responsibilities into multiple classes.
- Explanation: Promotes Single Responsibility Principle (SRP).



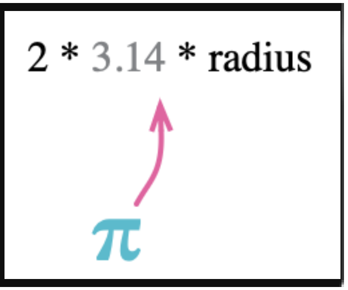
```python
class Person:
    def __init__(self, name, telephone_number):
        self.name = name
        self._telephone_number = telephone_number

    def get_telephone_number(self):
        return self._telephone_number

class TelephoneNumber:
    def __init__(self, office_area_code, office_number):
        self._office_area_code = office_area_code
        self._office_number = office_number

    def get_telephone_number(self):
        return '(' + self._office_area_code + ') ' + self._office_number
```

Example 6: Replace Magic Number with Symbolic Constant

✧ Before: Hard-coded numbers in code.



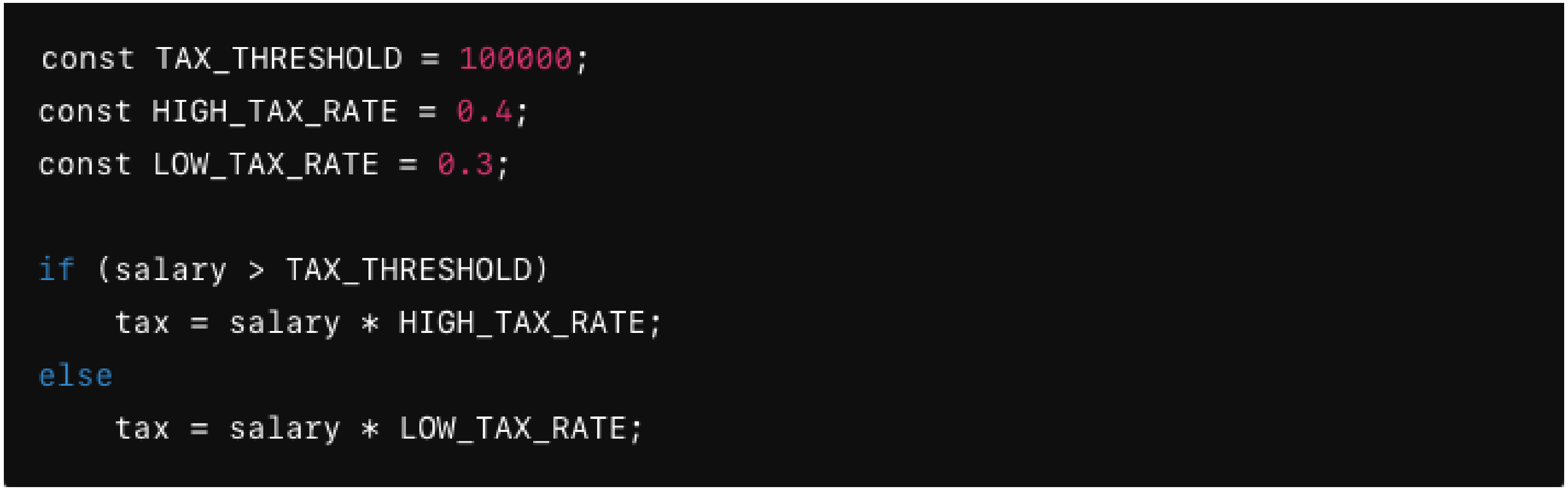
```
if (salary > 100000)
    tax = salary * 0.4;
else
    tax = salary * 0.3;
```



Example 6: Replace Magic Number with Symbolic Constant

✧ After: Replace with named constants.

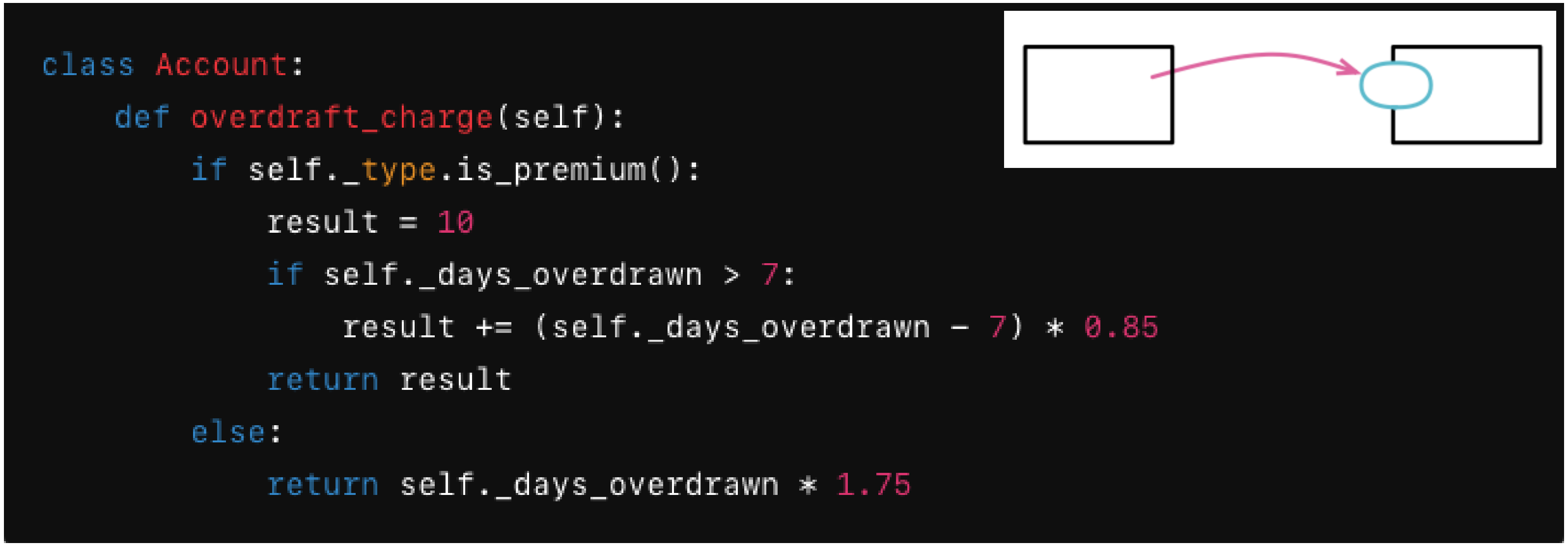
✧ Explanation: Makes the code self-explanatory.



```
const TAX_THRESHOLD = 100000;
const HIGH_TAX_RATE = 0.4;
const LOW_TAX_RATE = 0.3;

if (salary > TAX_THRESHOLD)
    tax = salary * HIGH_TAX_RATE;
else
    tax = salary * LOW_TAX_RATE;
```

Example 7: Move Method

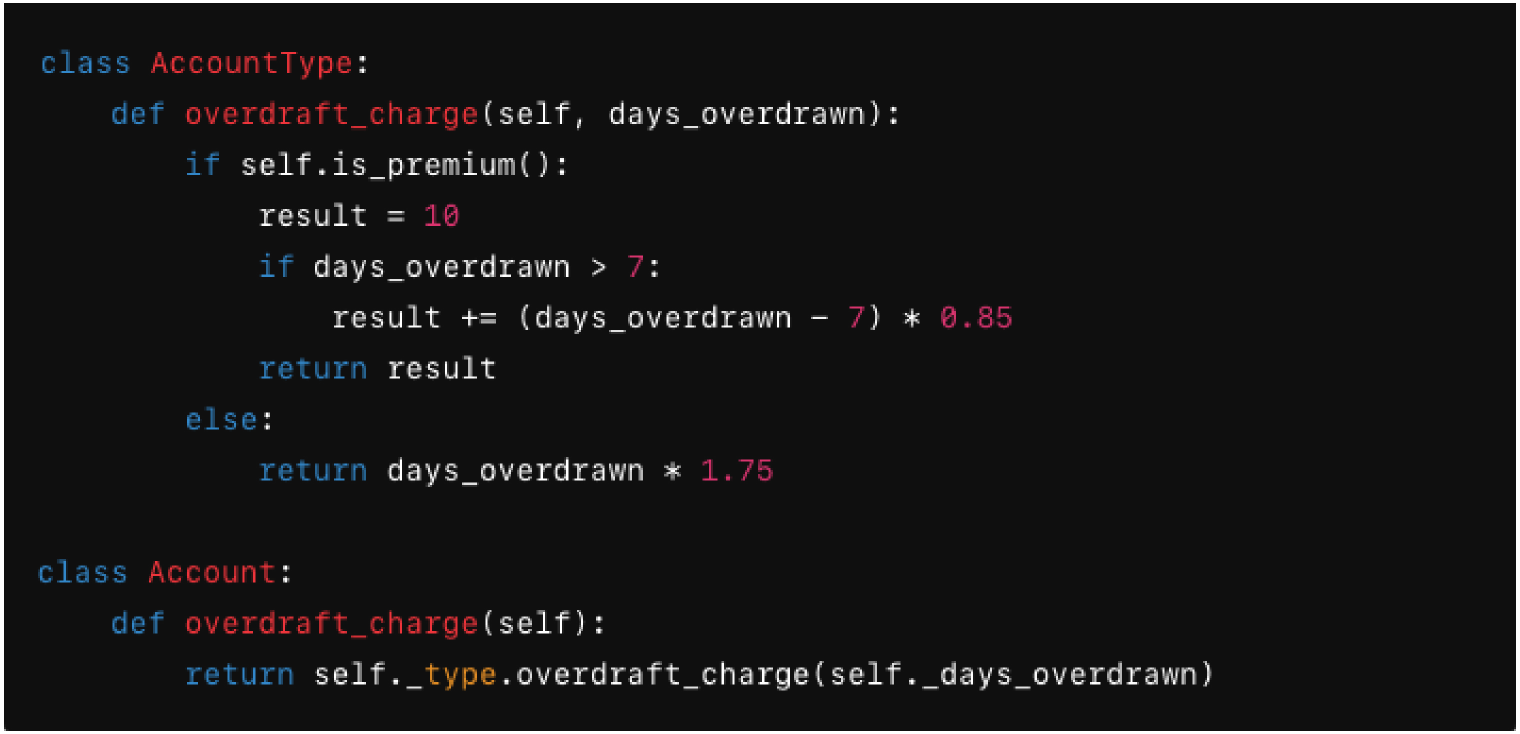
✧ Before: Method more related to another class.



```python
class Account:
    def overdraft_charge(self):
        if self._type.is_premium():
            result = 10
            if self._days_overdrawn > 7:
                result += (self._days_overdrawn - 7) * 0.85
            return result
        else:
            return self._days_overdrawn * 1.75
```

Example 7: Move Method

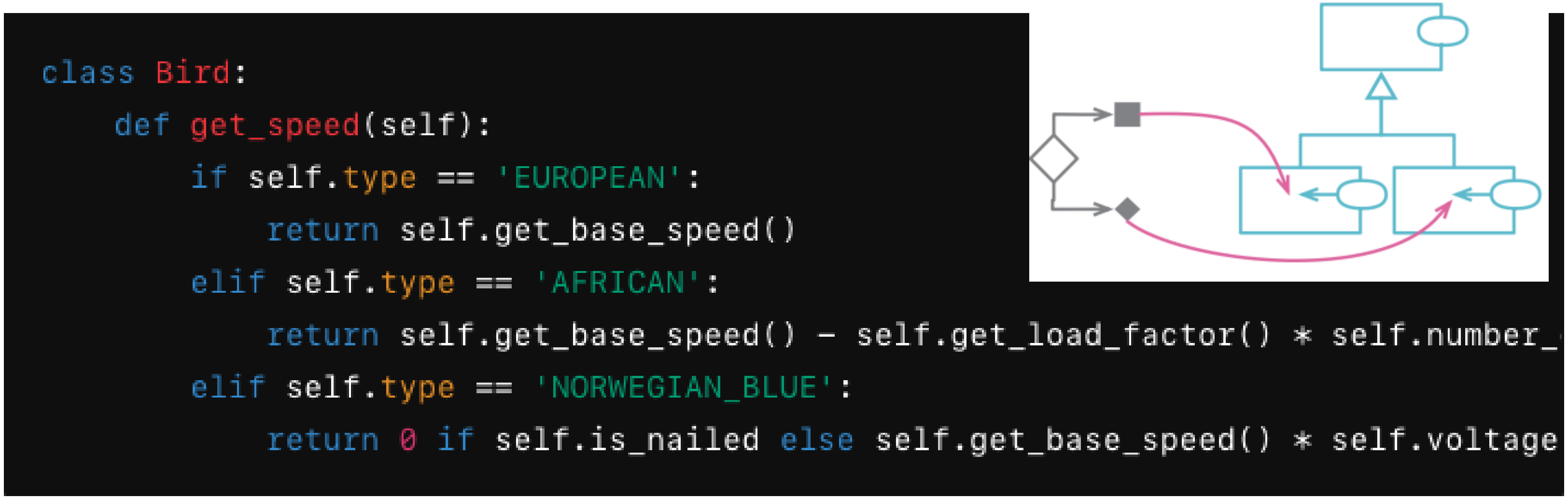
- After: Move the method to appropriate class.
- Explanation: Improves code organization and relevance.



```
class AccountType:
    def overdraft_charge(self, days_overdrawn):
        if self.is_premium():
            result = 10
            if days_overdrawn > 7:
                result += (days_overdrawn - 7) * 0.85
            return result
        else:
            return days_overdrawn * 1.75

class Account:
    def overdraft_charge(self):
        return self._type.overdraft_charge(self._days_overdrawn)
```

Example 8: Replace Conditional with Polymorphism

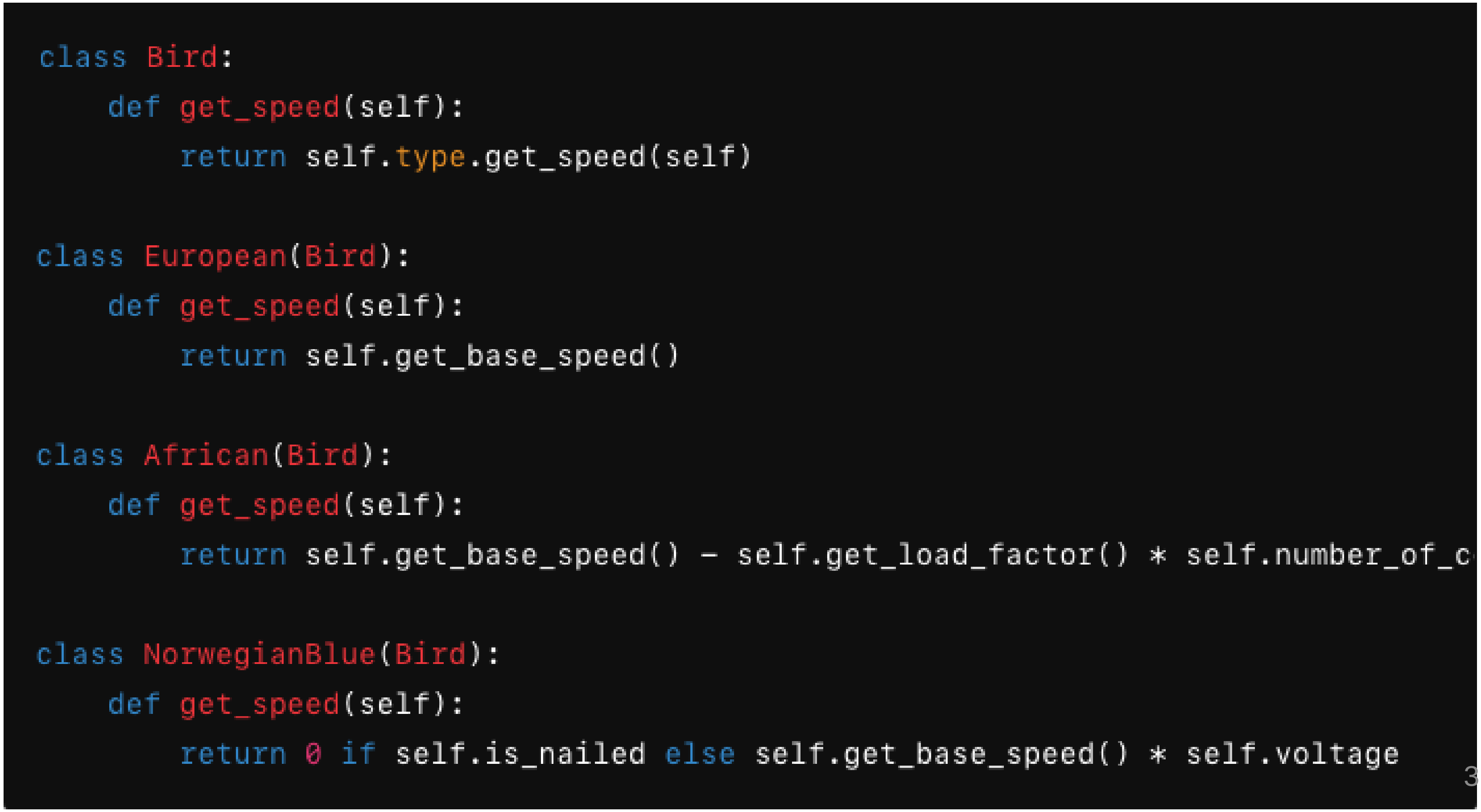
✧ Before: Complex conditional logic.



```
class Bird:
    def get_speed(self):
        if self.type == 'EUROPEAN':
            return self.get_base_speed()
        elif self.type == 'AFRICAN':
            return self.get_base_speed() - self.get_load_factor() * self.number_
        elif self.type == 'NORWEGIAN_BLUE':
            return 0 if self.is_nailed else self.get_base_speed() * self.voltage
```

Example 8: Replace Conditional with Polymorphism

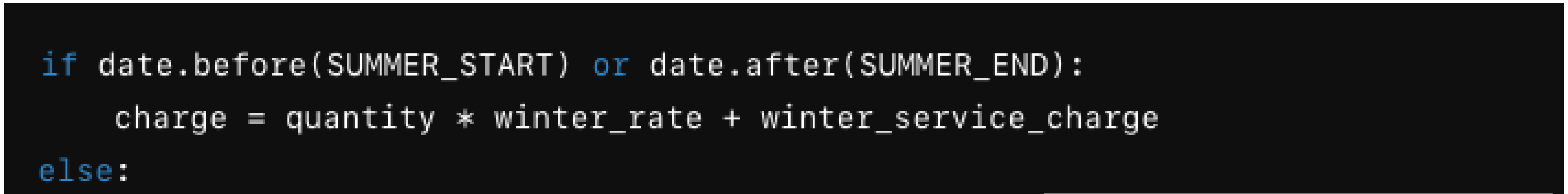
- After: Replace with polymorphic classes or strategies.
- Explanation: Simplifies logic and enhances flexibility.



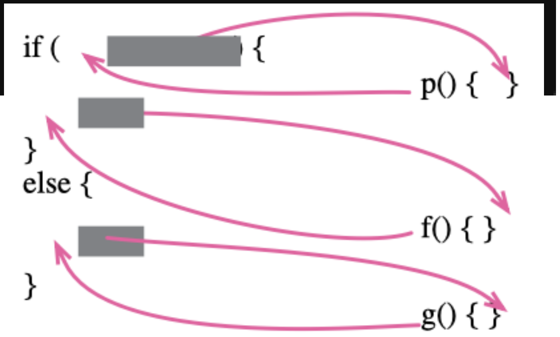
```
class Bird:
    def get_speed(self):
        return self.type.get_speed(self)

class European(Bird):
    def get_speed(self):
        return self.get_base_speed()

class African(Bird):
    def get_speed(self):
        return self.get_base_speed() - self.get_load_factor() * self.number_of_c

class NorwegianBlue(Bird):
    def get_speed(self):
        return 0 if self.is_nailed else self.get_base_speed() * self.voltage
```

Example 9: Decompose Conditional

✧ Before: Complex and nested conditional statements

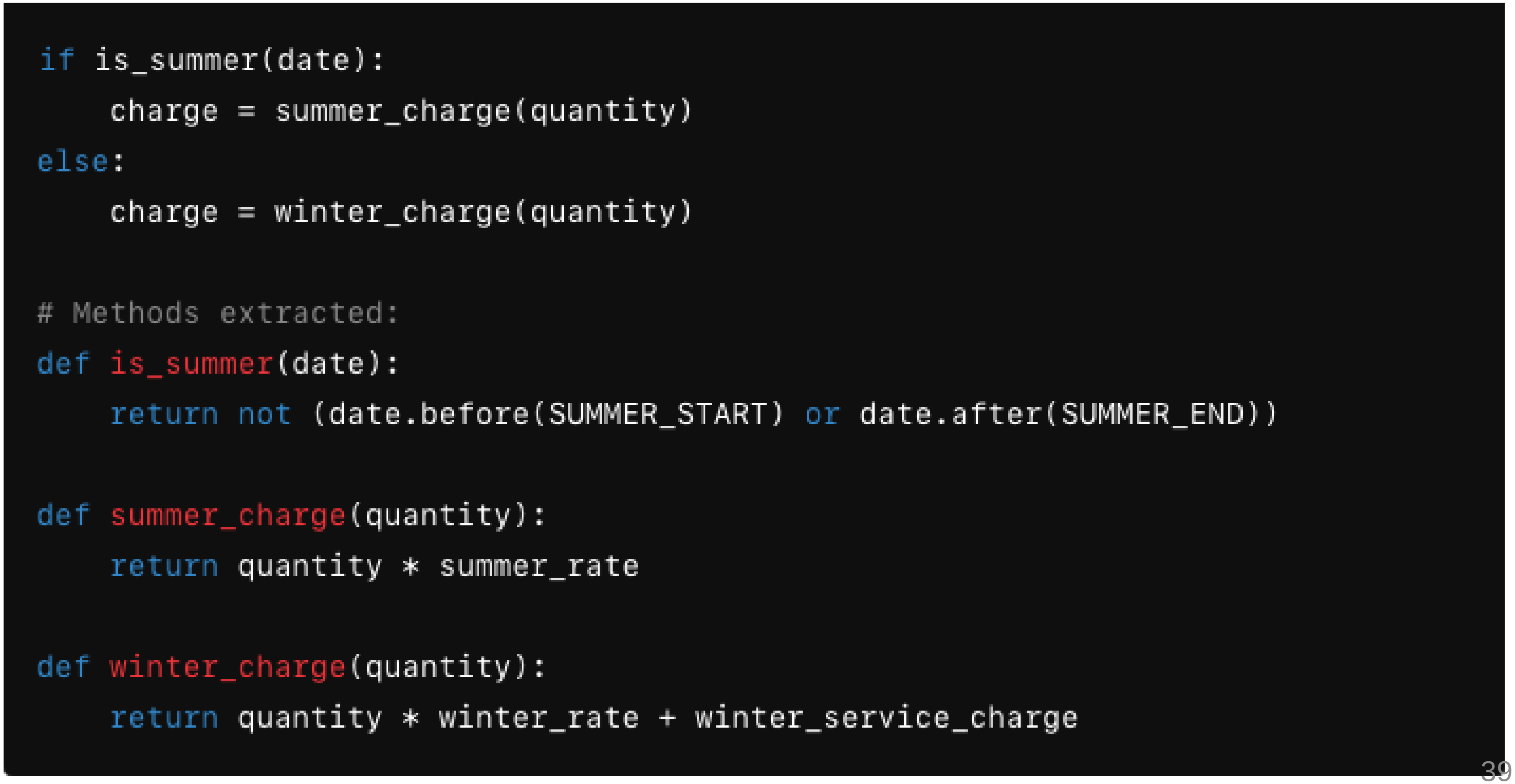


```
if date.before(SUMMER_START) or date.after(SUMMER_END):
    charge = quantity * winter_rate + winter_service_charge
else:
    charge = quantity * summer_rate
```



Example 9: Decompose Conditional

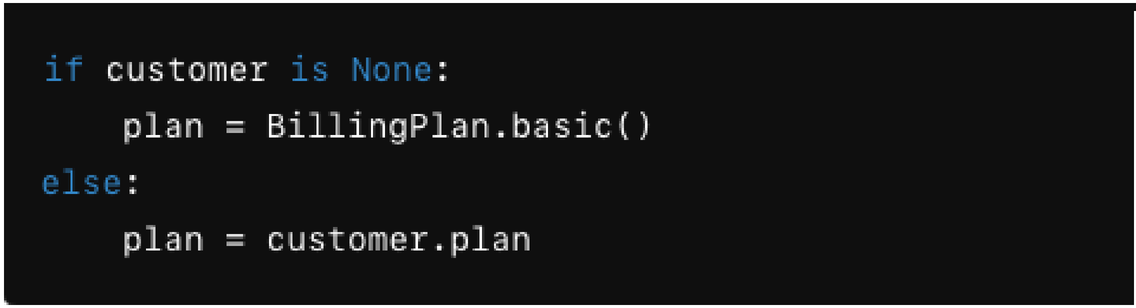
- After: Break down into methods with clear names.
- Explanation: Increases clarity and reduces code complexity.



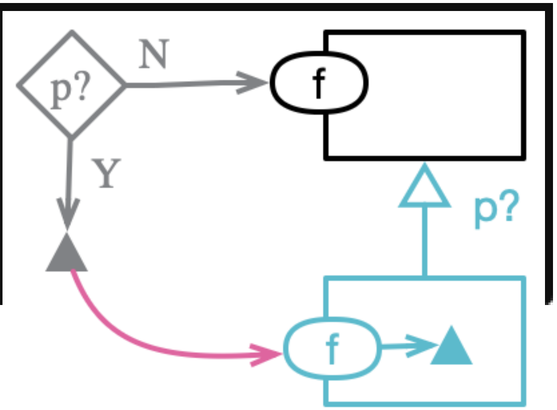
```
if is_summer(date):
    charge = summer_charge(quantity)
else:
    charge = winter_charge(quantity)

# Methods extracted:
def is_summer(date):
    return not (date.before(SUMMER_START) or date.after(SUMMER_END))

def summer_charge(quantity):
    return quantity * summer_rate

def winter_charge(quantity):
    return quantity * winter_rate + winter_service_charge
```

Example 10: Introduce Null Object

✧ Before: Null checks scattered throughout the code.

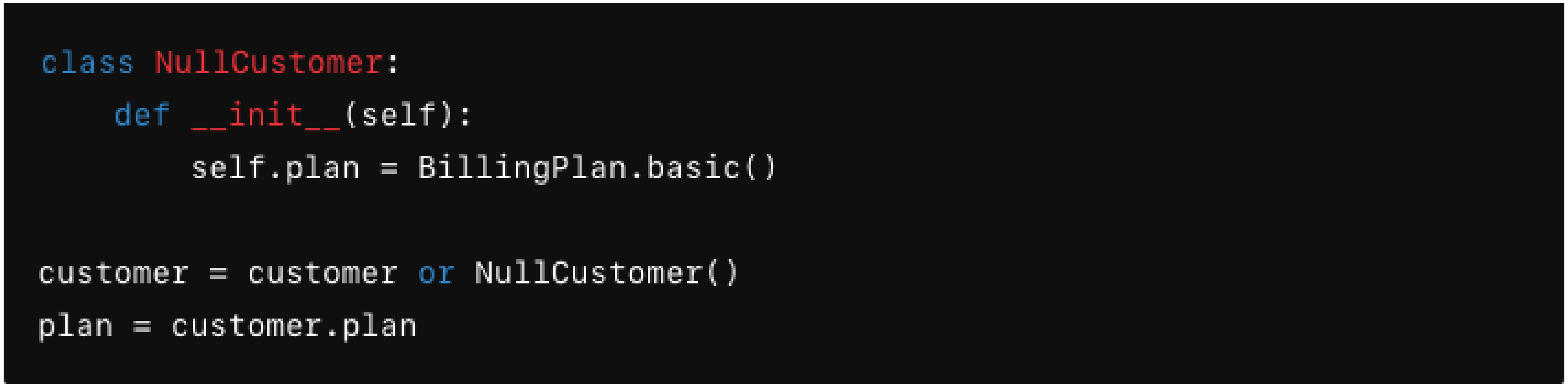


```
if customer is None:
    plan = BillingPlan.basic()
else:
    plan = customer.plan
```



Example 10: Introduce Null Object

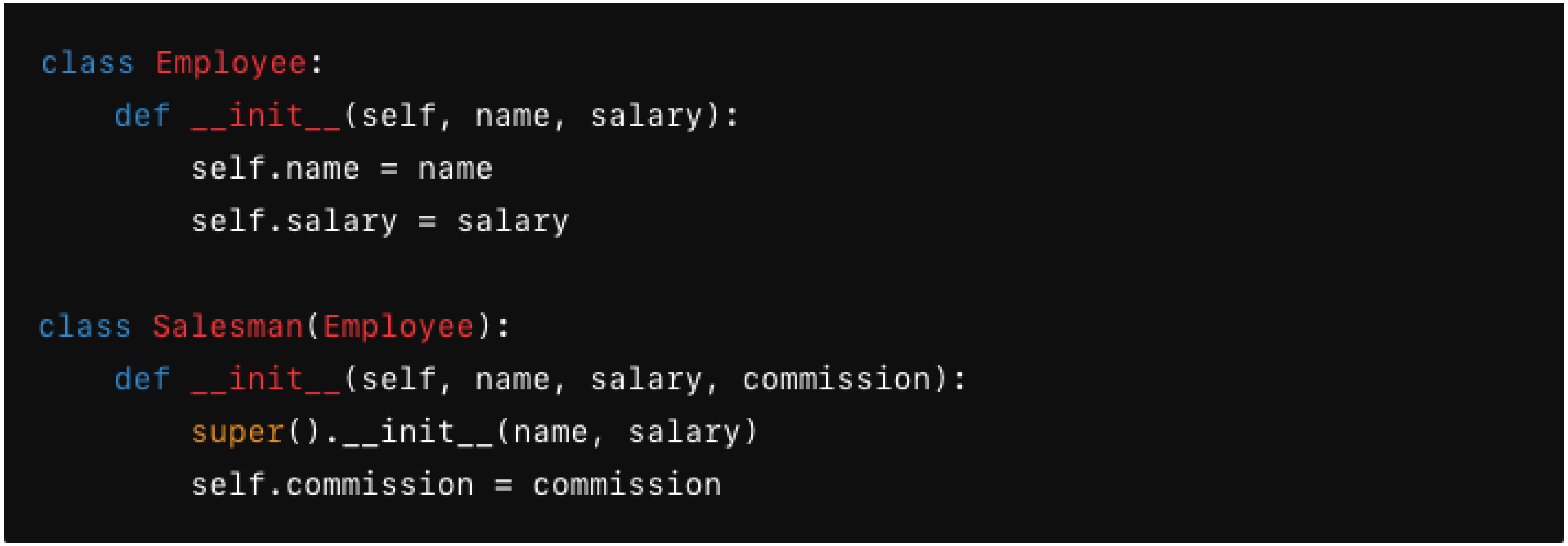
- ✧ After: Introduce a Null Object to represent absence of an object.
- ✧ Explanation: Simplifies code by removing null checks.



```python
class NullCustomer:
    def __init__(self):
        self.plan = BillingPlan.basic()

customer = customer or NullCustomer()
plan = customer.plan
```

Example 11: Replace Inheritance with Composition

- Before: Inheritance leads to rigid and brittle code.

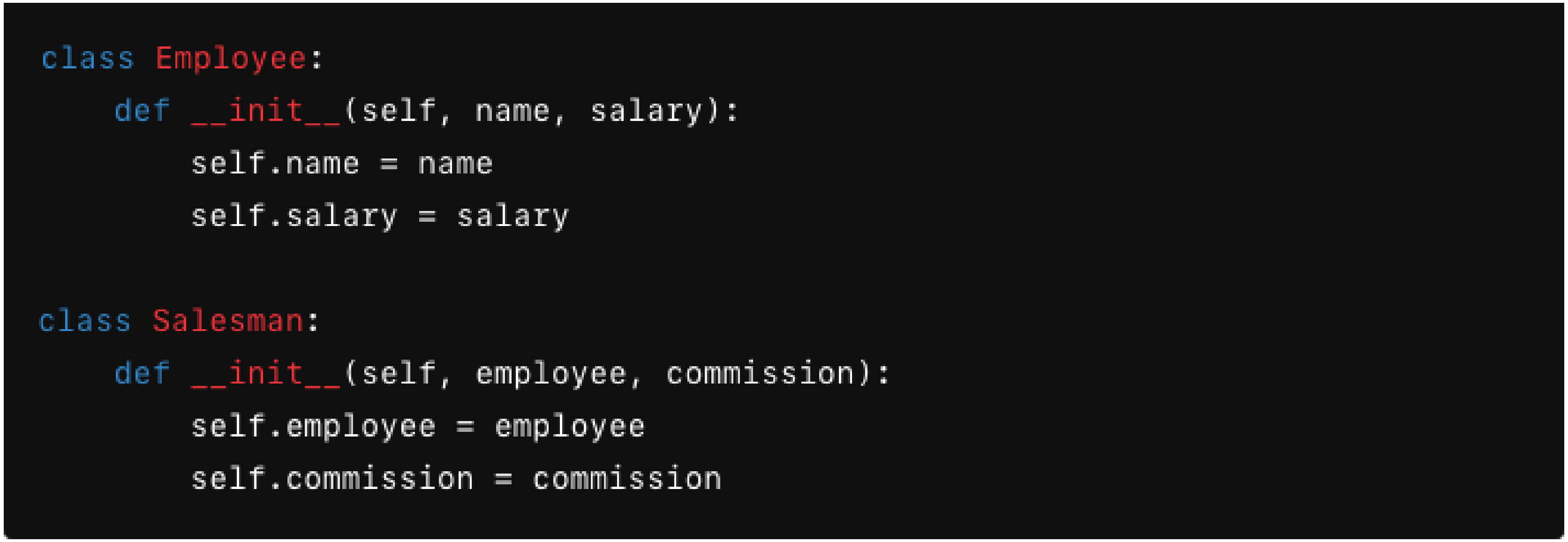


```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Salesman(Employee):
    def __init__(self, name, salary, commission):
        super().__init__(name, salary)
        self.commission = commission
```

Example 11: Replace Inheritance with Composition

✧ After: Use composition instead of inheritance.

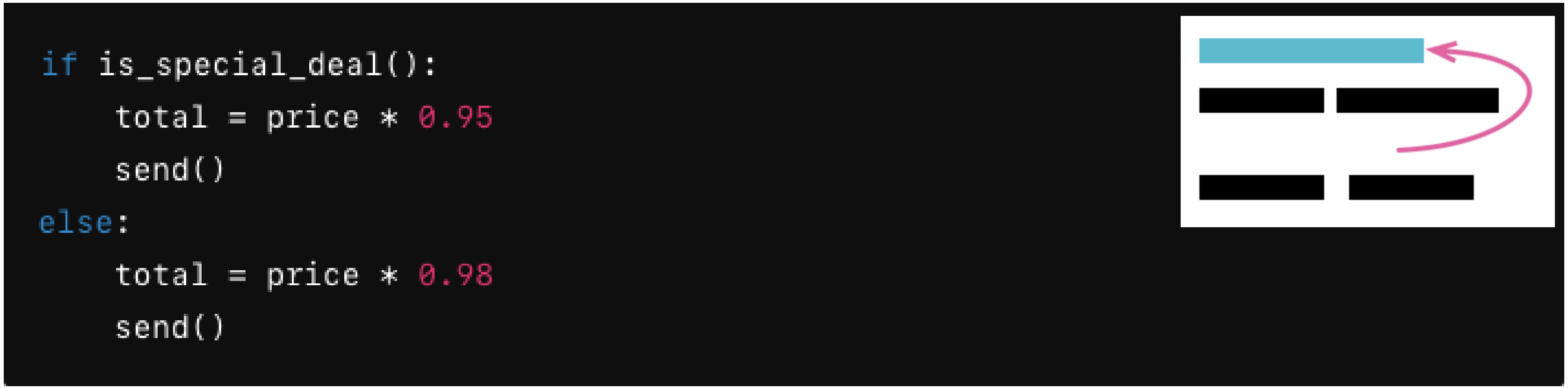
✧ Explanation: Improves flexibility and reusability.



```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Salesman:
    def __init__(self, employee, commission):
        self.employee = employee
        self.commission = commission
```

Example 12: Consolidate Duplicate Conditional Fragments

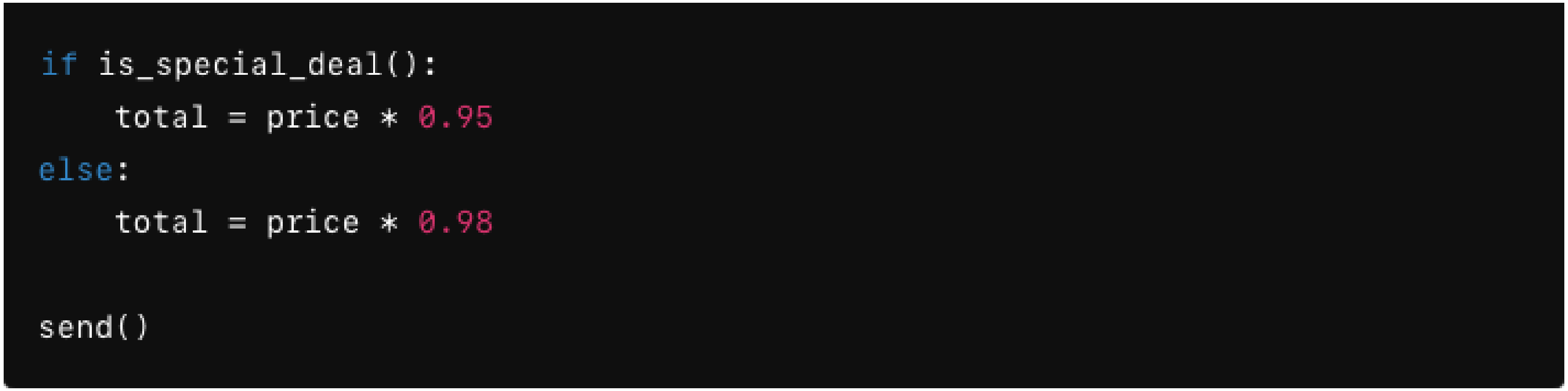
✧ Before: Duplicate code within conditional branches.



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

Example 12: Consolidate Duplicate Conditional Fragments

- After: Consolidate duplicate code outside the conditional.
- Explanation: Reduces redundancy and enhances maintainability.



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

send()
```

Dangers of Refactoring

- Code that used to be well commented, well tested, and fully reviewed might not be any of these things after refactoring.
- You might have inserted faults into code that previously worked.
  - This is why unit tests are important. If the new code is broken, revert back to the old code.
- What if the new design is not better?

“I Don’t Have Time”

No thanks!
We are too busy.

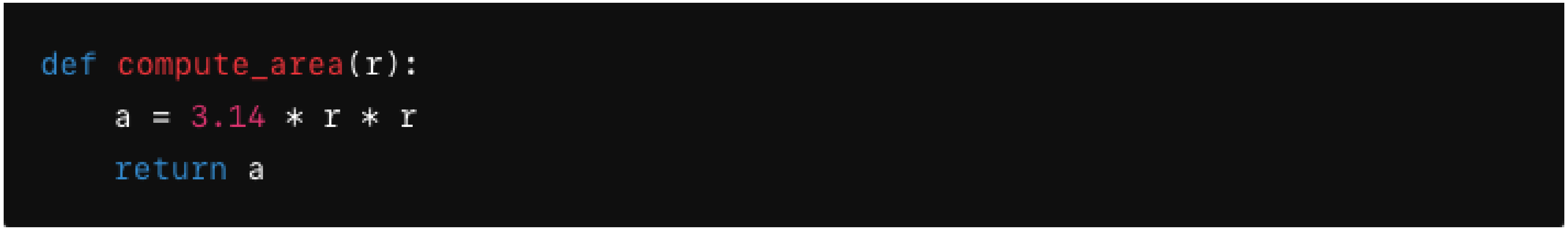
“I Don’t Have Time”

- Most common excuse for not refactoring.
- Refactoring incurs an up-front cost.
  - Developers don’t want to do it.
  - Neither do managers - they lose time and get “nothing” (no new features)
- Small companies (start-ups) avoid it.
  - “We can’t afford it.” “We don’t need it.”
- So do large companies.
  - “We’d rather add new features.”
  - “No one gets promoted for refactoring.”

“I Don’t Have Time”

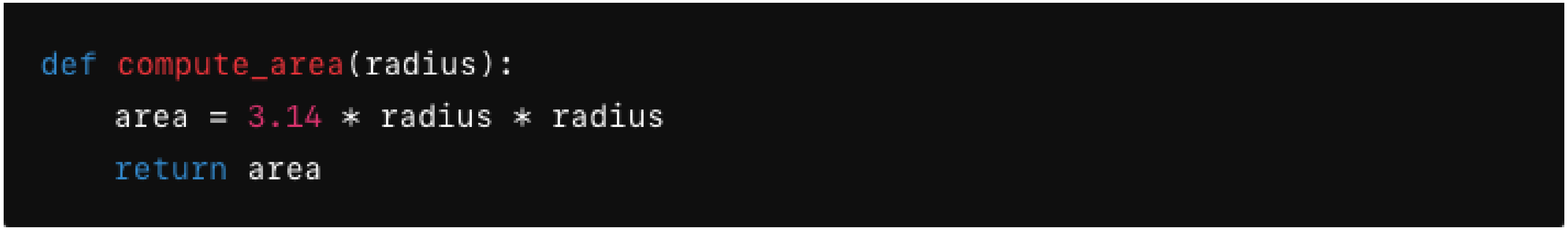
- Refactoring is the key to effective evolution.
    - Enables rapid addition of new features, with fewer faults (up to a 500% ROI).
    - Good for programmer morale.
- Refactoring is an investment in a company’s prime asset - its code base.
- Many start-ups use cutting-edge tech and agile processes that evolve rapidly. So should the code.
- Some of the most successful companies (Google) reward and require refactoring.

Practice#1: Identify the *code smell* and suggest a possible refactoring



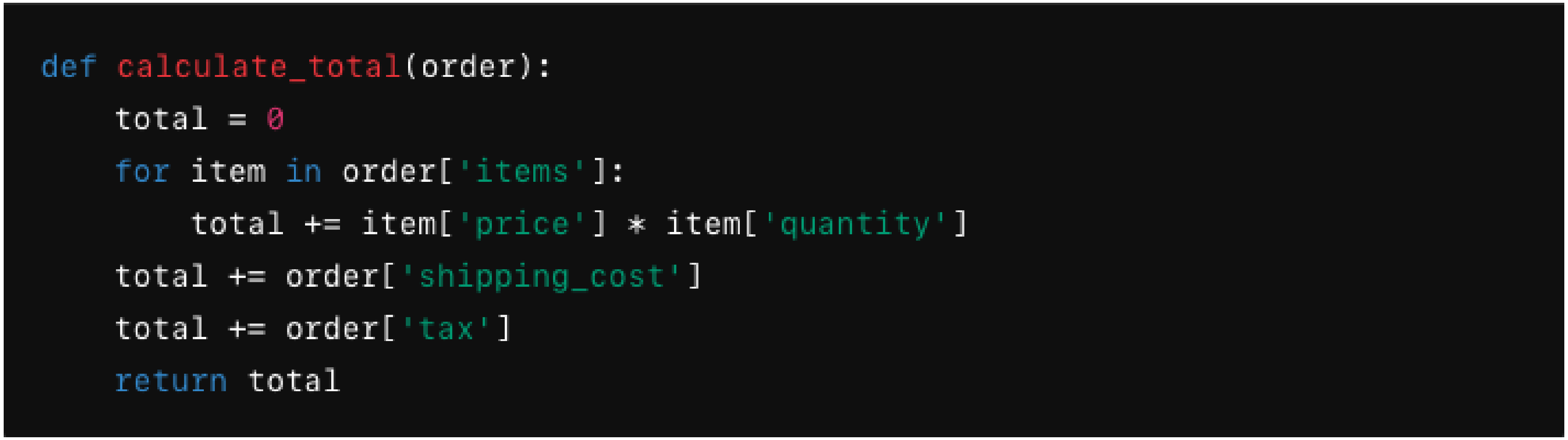
```
def compute_area(r):
    a = 3.14 * r * r
    return a
```

Practice#1: Solution (Rename Variable)



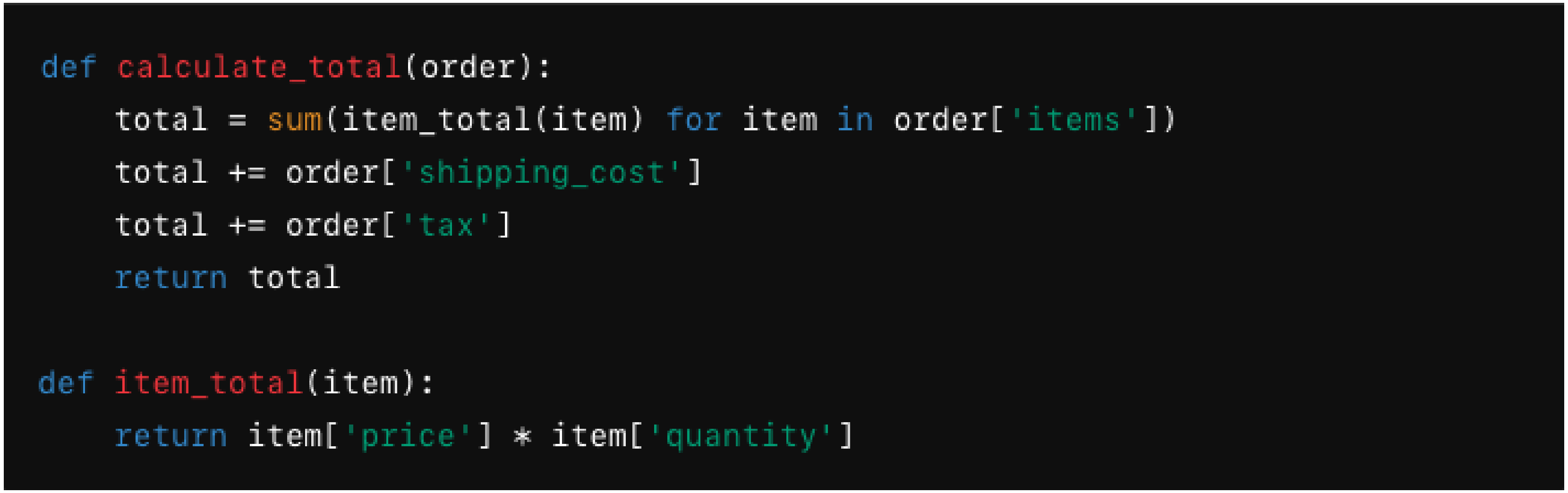
```
def compute_area(radius):
    area = 3.14 * radius * radius
    return area
```

Practice#2: Identify the *code smell* and suggest a possible refactoring



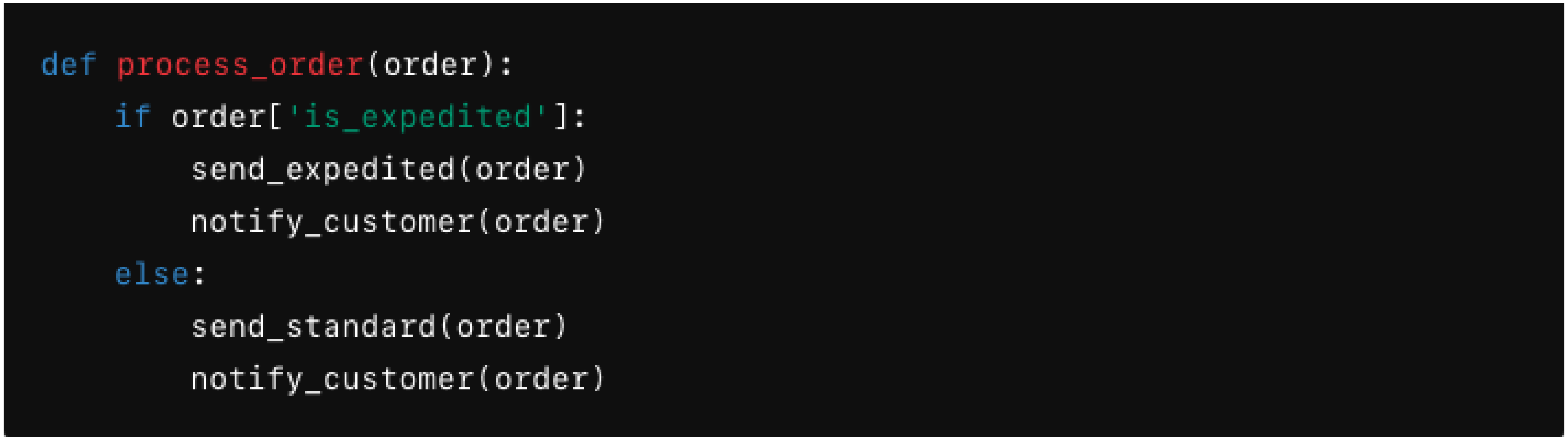
```python
def calculate_total(order):
    total = 0
    for item in order['items']:
        total += item['price'] * item['quantity']
    total += order['shipping_cost']
    total += order['tax']
    return total
```

Practice#2: Solution (Extract Method)



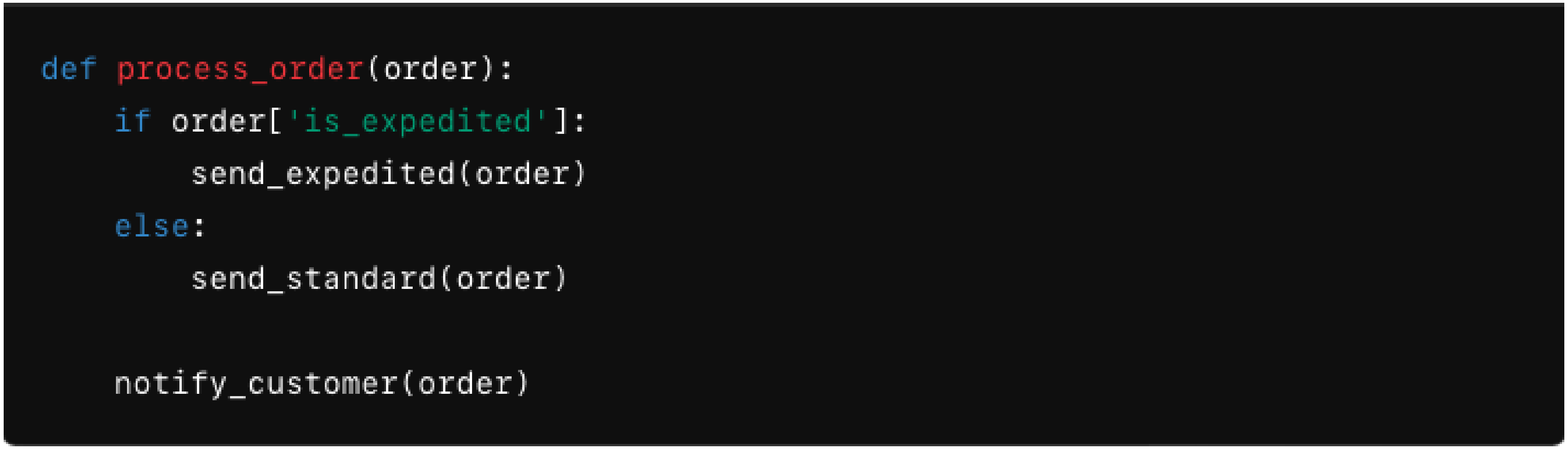
```python
def calculate_total(order):
    total = sum(item_total(item) for item in order['items'])
    total += order['shipping_cost']
    total += order['tax']
    return total

def item_total(item):
    return item['price'] * item['quantity']
```

Practice#3: Identify the *code smell* and suggest a possible refactoring



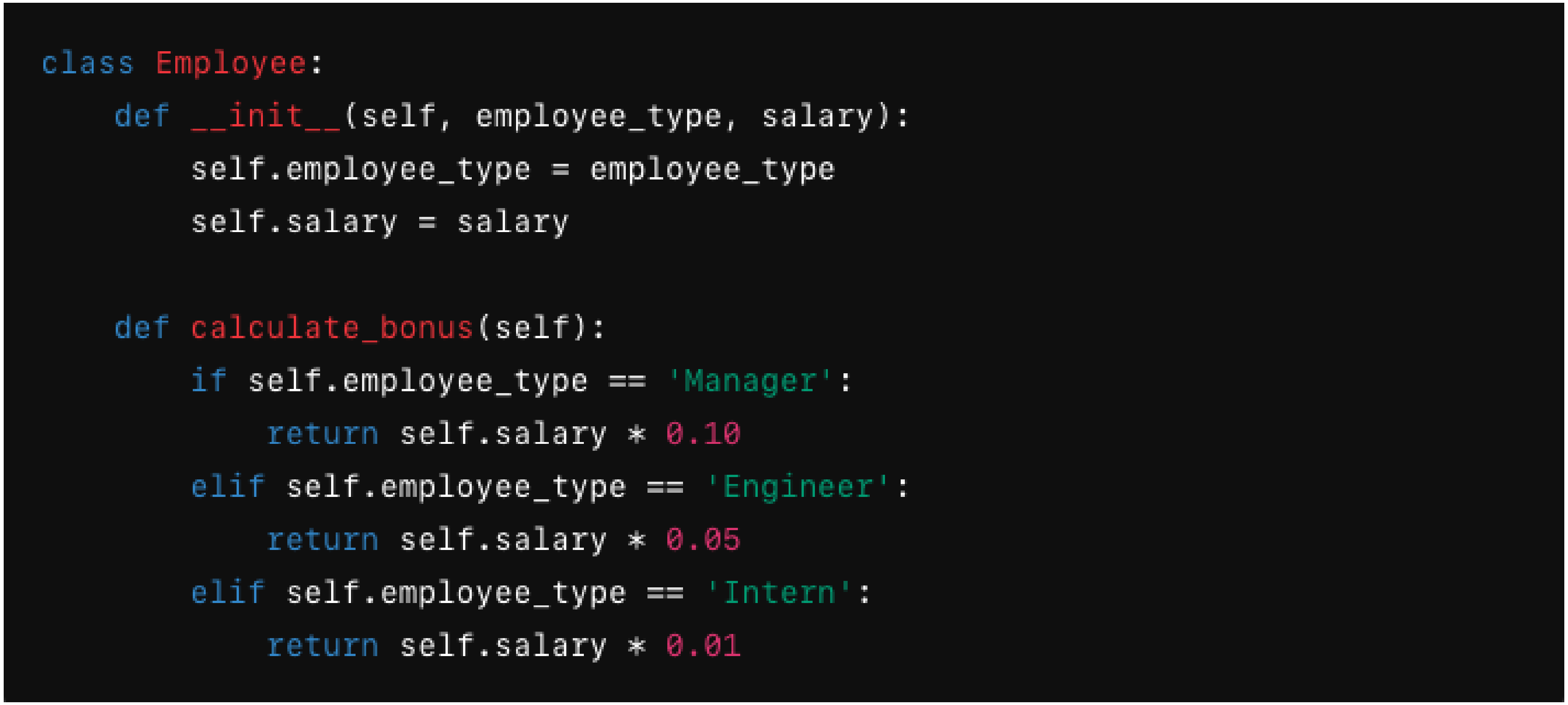
```python
def process_order(order):
    if order['is_expedited']:
        send_expedited(order)
        notify_customer(order)
    else:
        send_standard(order)
        notify_customer(order)
```

Practice#3: Solution (Consolidate Duplicate Conditional Fragments)



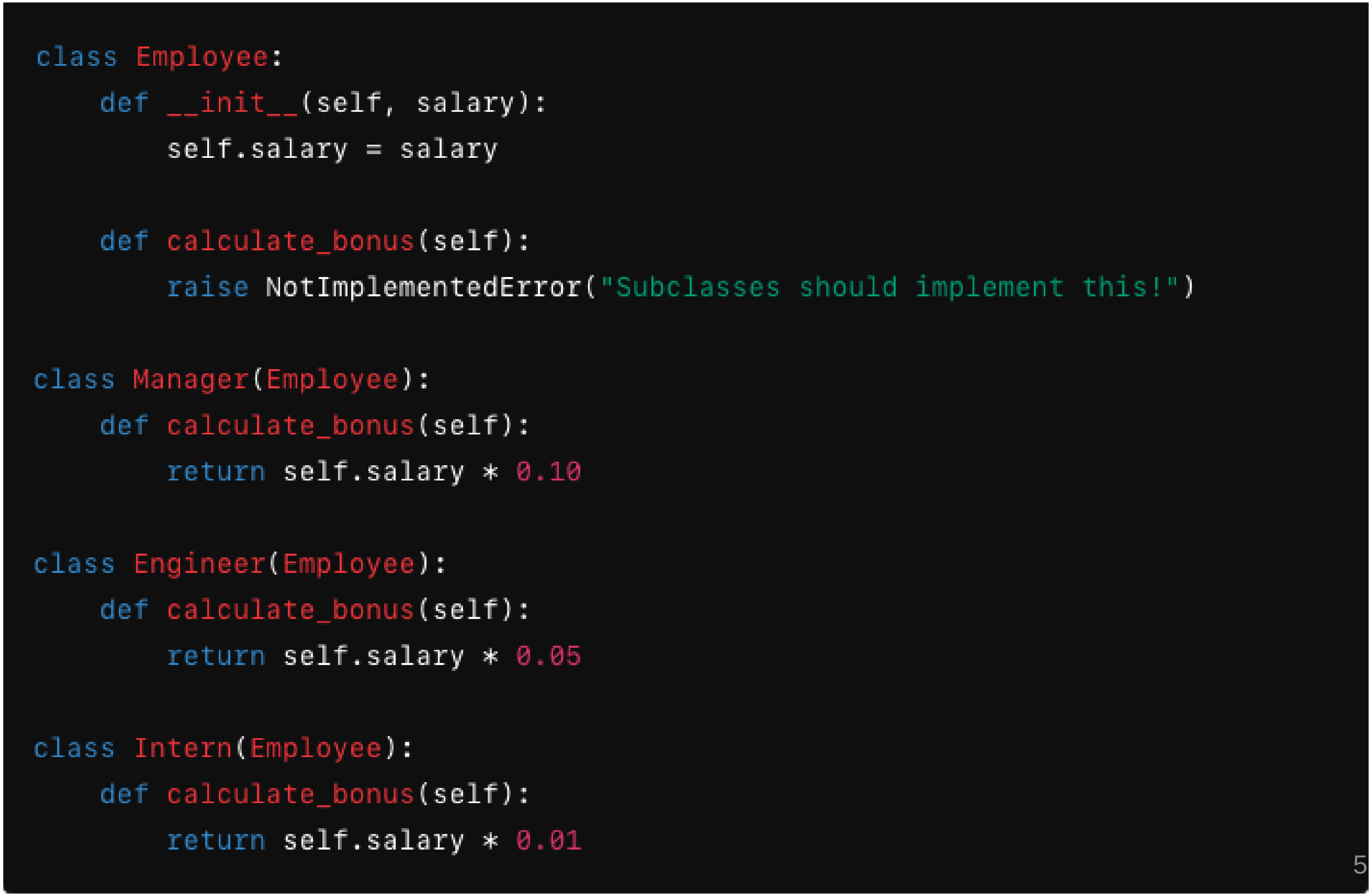
```
def process_order(order):
    if order['is_expedited']:
        send_expedited(order)
    else:
        send_standard(order)

    notify_customer(order)
```

Practice#4: Identify the *code smell* and suggest a possible refactoring



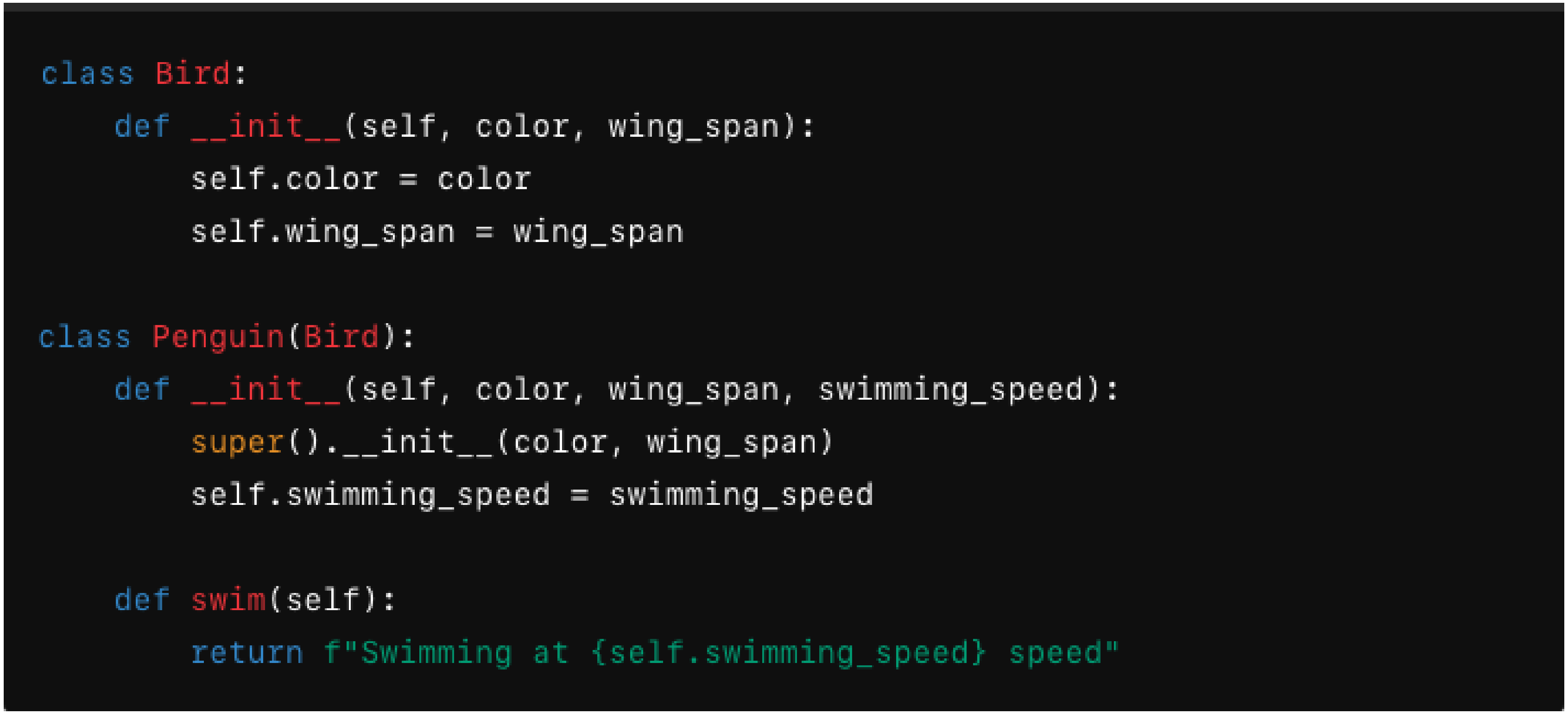
```python
class Employee:
    def __init__(self, employee_type, salary):
        self.employee_type = employee_type
        self.salary = salary

    def calculate_bonus(self):
        if self.employee_type == 'Manager':
            return self.salary * 0.10
        elif self.employee_type == 'Engineer':
            return self.salary * 0.05
        elif self.employee_type == 'Intern':
            return self.salary * 0.01
```

Practice#4: Solution (Replace Conditional with Polymorphism)



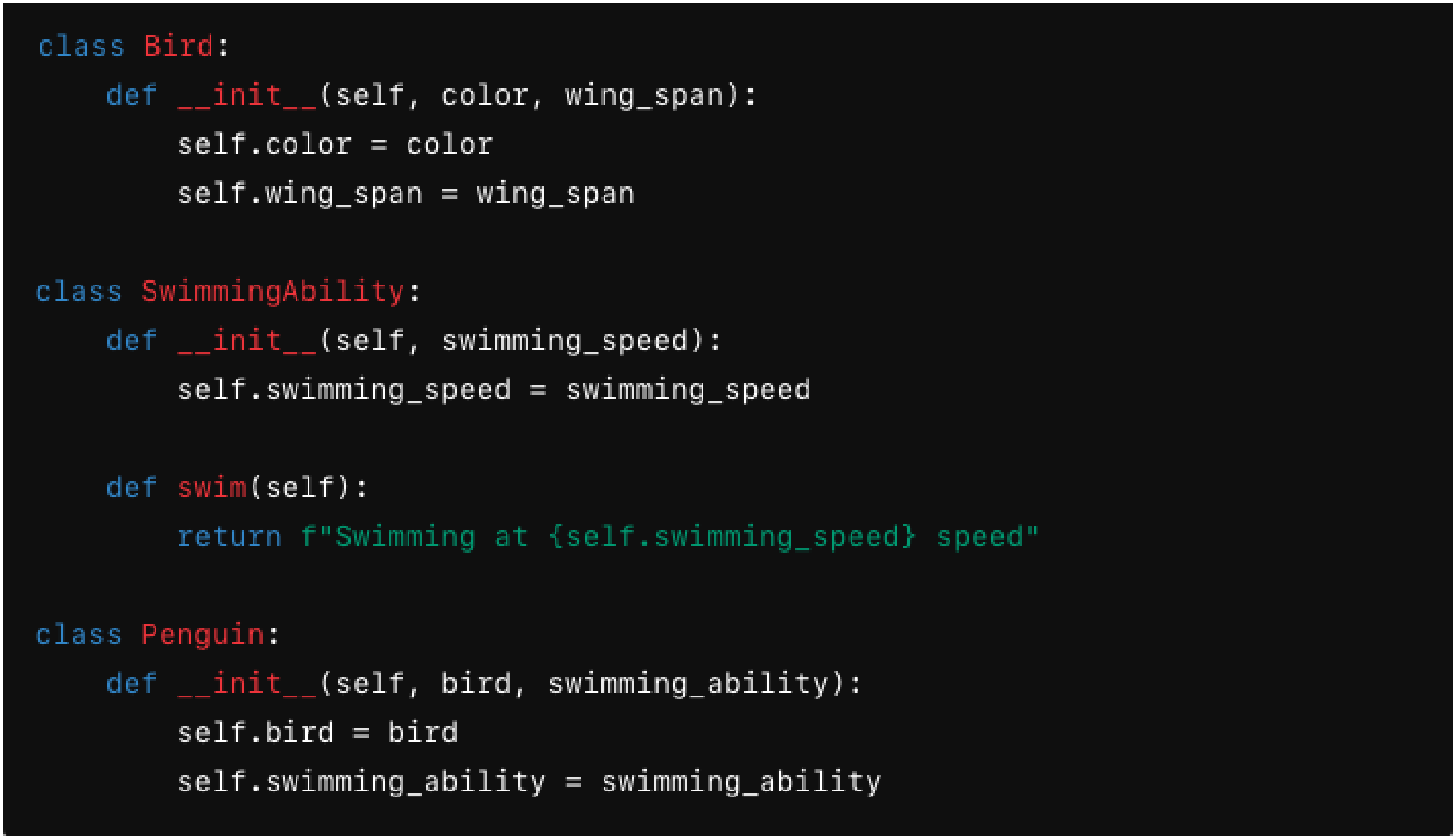
```python
class Employee:
    def __init__(self, salary):
        self.salary = salary

    def calculate_bonus(self):
        raise NotImplementedError("Subclasses should implement this!")

class Manager(Employee):
    def calculate_bonus(self):
        return self.salary * 0.10

class Engineer(Employee):
    def calculate_bonus(self):
        return self.salary * 0.05

class Intern(Employee):
    def calculate_bonus(self):
        return self.salary * 0.01
```

Practice#5: Identify the *code smell* and suggest a possible refactoring



```python
class Bird:
    def __init__(self, color, wing_span):
        self.color = color
        self.wing_span = wing_span

class Penguin(Bird):
    def __init__(self, color, wing_span, swimming_speed):
        super().__init__(color, wing_span)
        self.swimming_speed = swimming_speed

    def swim(self):
        return f"Swimming at {self.swimming_speed} speed"
```

Practice#5: Solution (Replace Inheritance with Composition)



```
class Bird:
    def __init__(self, color, wing_span):
        self.color = color
        self.wing_span = wing_span

class SwimmingAbility:
    def __init__(self, swimming_speed):
        self.swimming_speed = swimming_speed

    def swim(self):
        return f"Swimming at {self.swimming_speed} speed"

class Penguin:
    def __init__(self, bird, swimming_ability):
        self.bird = bird
        self.swimming_ability = swimming_ability
```

HOW
WHY
HOW
WHEN
WHERE
WHO
WHERE
WHERE
WHY
WHO
WHO
WHEN
WHAT
WHAT
WHY
WHAT
WHEN
HOW
WHAT
WHAT
WHEN
WHO
WHERE
WHY
HOW
?