JAVA-201

Code Smells

Code Smells

"Mabahong code."

"A code smell is a surface indication that usually corresponds to a deeper problem in the system." -- Martin Fowler

Providing names to different types of ugly code enables more effective discussions on design improvement.

Here are some of the more well-known code smells...

Duplicate Behavior ("Duplicate Code")

How does it happen?

- Code works... you need the same behavior somewhere else... copy... paste.
- Repeat N times.
  - Maybe some slight variations in each spot. Everything's fine... until you need to make changes...
- You find all the spots where you copy-pasted code, and then you make edits.
- Maybe you forget one or more spots... bugs appear... you find, fix, retest, find, fix...

Duplicate Behavior ("Duplicate Code")

Then you need to make even more changes... and your code base grows. Oh, and you did copy-paste for a hundred other routines.

- You find yourself in a never-ending cycle of updating little spots of code all over the codebase.
- The routines get complex and difficult to understand.
- You can't be reassigned to a new project since you're the only one who understands the code.

Duplicate Behavior ("Duplicate Code")

```
class TotalCalculator:
    @staticmethod
    def calculate(items):
        total = 0
        for item in items:
            total += item['price'] * item['quantity']
        return total

class Invoice:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        return TotalCalculator.calculate(self.items)

class Order:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        return TotalCalculator.calculate(self.items)
```

Duplicate Behavior ("Duplicate Code")

Solution

Don’t Repeat Yourself (DRY)

Write code in a method just once, then call it where you need it.

- Use Composition or Inheritance to reduce duplication
- Use parameters or polymorphism to handle variations.

When you need to update your logic, you only need to update and test one spot.

Code is easier to read since it is broken up into descriptive methods.

- You used descriptive method names, right?

Patterns: Utility/Helper Class, Template Method, Strategy, State, Layer Supertype

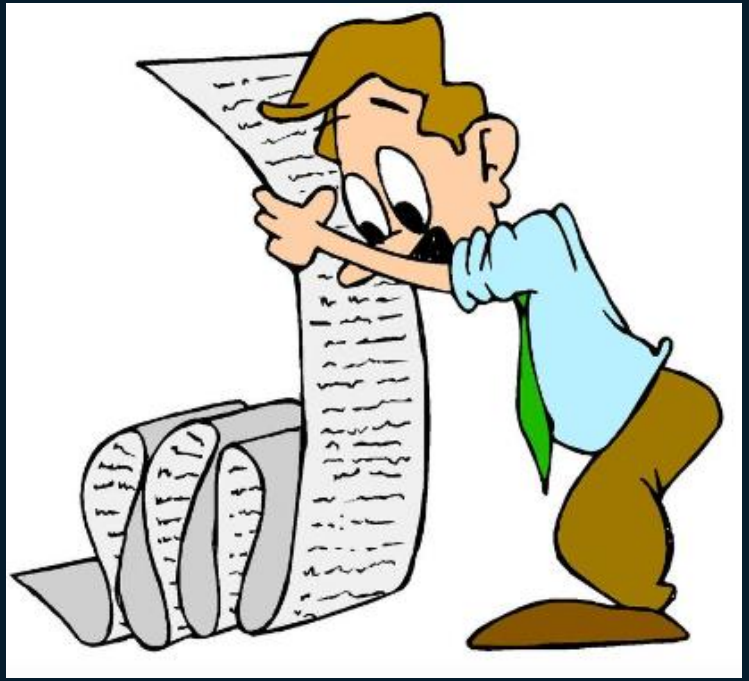
Long Methods & Large Classes

"Hmm, let me try to understand the method that my teammate wrote..." ... scroll... scroll... page-down... page-down...

Another warning sign: Class has a lot of fields.

Difficult to understand, maintain, reuse.

OOP is about small, specialized components.



Long Methods & Large Classes

Each method should only do one, simple thing.

- No side-effects.
- Method name should be descriptive of that one thing.
  - If a method is hard to name, it's probably doing to much.

A class should be a specialized, cohesive unit.

- Just a few fields.
- If the class is difficult to name, or has a generic name, it might be doing too much (or not enough).

Long Methods & Large Classes

```
class EmployeeManager:
    def add_employee(self, name, role, salary):
        print(f"Adding employee {name} as {role} with salary
${salary}")

    def calculate_payroll(self, employees):
        payroll = sum(employee['salary'] for employee in employees)
        print(f"Total payroll is ${payroll}")

    def generate_report(self, employees):
        for employee in employees:
            print(f"{employee['name']}: {employee['role']},
${employee['salary']}")
```

Long Methods & Large Classes

Solutions:

Break up a long method into multiple smaller methods.

Break up a large class into two or more classes.

Apply SRP, Information Expert, Law of Demeter

Primitive Obsession, Data Clumps & Long Parameter Lists

Primitive Obsession: Sticking with built-in types of the language instead of creating one's own classes.

```
int customerId1;
String customerFirstName1;
String customerLastName1;
int[] orderIdsOfCustomer1;
int customerId2;
String customerFirstName2;
String customerLastName2;
int[] orderIdsOfCustomer2;
int customerId3;
String customerFirstName3;
String customerLastName3;
int[] orderIdsOfCustomer3;
...
```

Primitive Obsession, Data Clumps & Long Parameter Lists

Data Clump: Data that's often found together.

```
int customerId;
String customerFirstName;
String customerLastName;
String customerEmailAddress;
int[] orderIdsOfCustomer;
Date[] orderDates;
...
```

Long Parameter Lists

```
void method(int customerId, int customerId, String customerFirstName,
            String customerLastName, String customerEmailAddress,
            int[] orderIdsOfCustomer, Date[] orderDates) {
```

Primitive Obsession, Data Clumps & Long Parameter Lists

What happens as the data clump changes?

- Additional form fields, removal of form fields, etc.
  - Remember, there was a time that when we did not need to ask people for their email addresses or mobile numbers, since most people had neither. Also in the past, most people had only one phone number. Also there was a time when it was common to ask for a person's pager number.

Maintenance nightmare!

Primitive Obsession, Data Clumps & Long Parameter Lists

Solution: **Cohesion** - Encapsulate a data clump into a meaningful class.

```
class Customer {
   private final int id;
   private String firstName;
   private String lastName;
   private String emailAddress;
   private Set<Order> orders;
   ...
}
```

Primitive Obsession, Data Clumps & Long Parameter Lists

Solution: **Cohesion** - Encapsulate a data clump into a meaningful class.

No more data clumps.

```
Customer customer1;
Customer customer2;
Customer customer3;
```

No more long parameter lists.

```
void method(Customer customer) {...}
```

Patterns: Rich Domain Model, State

Long Parameter Lists

Long parameter lists can also be a sign that the method is in the wrong class.

- It always requires data from somewhere else.

OOP is about combining data structures with the operations that act on them.

Solution: Move the method to the class where most or all of the data it uses is found.

Apply Information Expert / Law of Demeter

Divergent Change

The same class is changed for different reasons.

"Well, I will have to change these three methods every time I get a new database and I have to change these four methods every time there is a new financial instrument.."

This class is handling both business and persistence concerns.
A class is suffering from divergent change if it breaks the principle of Separation of Concerns.

Divergent Change

```
class DocumentManager:
    def format_as_pdf(self, content):
        print(f"Formatting content as PDF: {content}")

    def format_as_word(self, content):
        print(f"Formatting content as Word: {content}")

    def save_to_file(self, content, file_path):
        with open(file_path, 'w') as file:
            file.write(content)
        print(f"Content saved to {file_path}")

    def print_document(self, content):
        print(f"Printing document: {content}")
```

Divergent Change
Solution

Similar to large classes, you should break up the class so that it only does one concern.

- Examples of concerns: business logic, persistence, presentation, transactions, security...

Shotgun Surgery

When multiple classes need to be modified to do one change.

- Opposite of Divergent Change.
- Caused by a lack of Cohesiveness - the many separate bits should be in one class.

Solution: **Cohesion** - Encapsulate the things that change together into one class.

"Things that change together should be together."
Also applies to packages - Classes that change together should be in the same package.

Shotgun Surgery

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class EmailService:
    def send_welcome_email(self, user):
        print(f"Sending welcome email to {user.email}...")

class LoggingService:
    def log_user_creation(self, user):
        print(f"User {user.name} has been created.")

class NotificationService:
    def send_account_notification(self, user):
        print(f"Sending account notification to {user.name}...")
```

```
# In a controller or main program:
class UserController:
    def create_user(self, name, email):
        user = User(name, email)
        EmailService().send_welcome_email(user)
        LoggingService().log_user_creation(user)
        NotificationService().send_account_notification(user)
```

Feature Envy

A method keeps accessing data from another class.

Solution: **Move the method** to the class where the data it uses resides.

- Again - OOP is combining data with the methods that act on that data

Apply Information Expert / Law of Demeter

Feature Envy

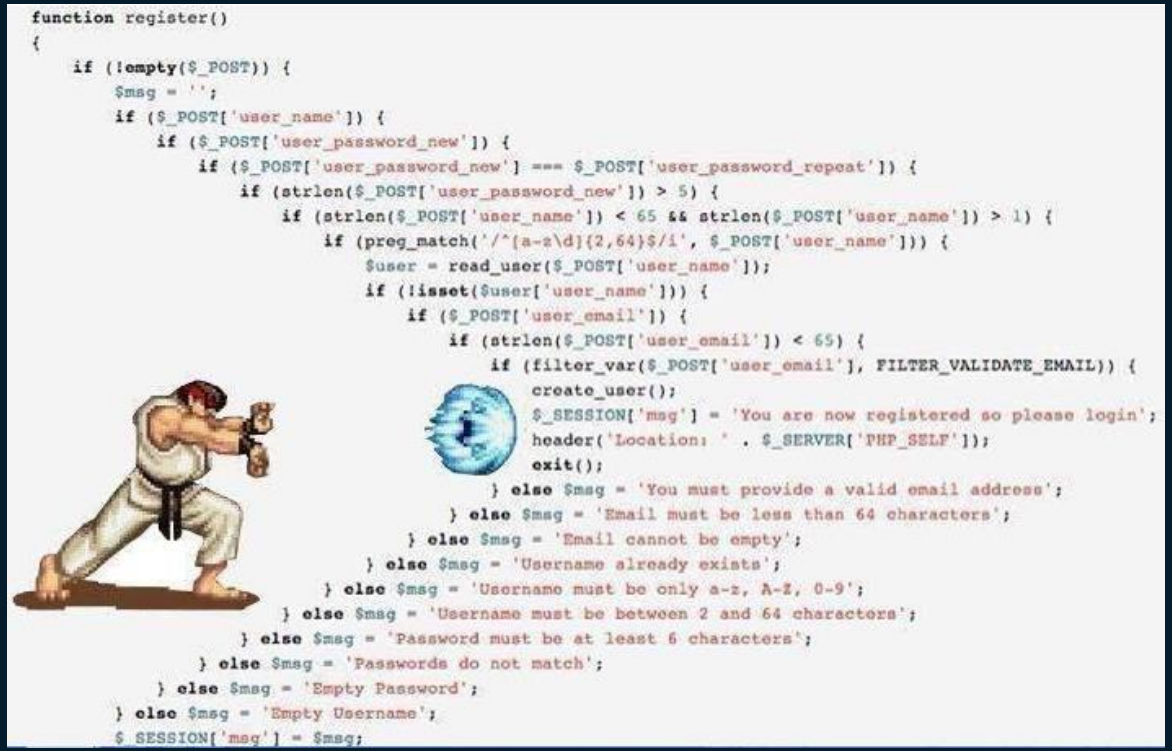
```
class Address:
    def __init__(self, street, city, zip_code):
        self.street = street
        self.city = city
        self.zip_code = zip_code

class Order:
    def __init__(self, order_id, address):
        self.order_id = order_id
        self.address = address

    def print_shipping_label(self):
        # This method heavily interacts with Address data.
        return f"Shipping to:\n{self.address.street}, {self.address.city}, {self.address.zip_code}"
```

Conditional Complexity

Large and growing If-Else and Switch logic become difficult to maintain.



```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
```

Conditional Complexity

```
class PaymentProcessor:
    def process_payment(self, payment_type, amount):
        if payment_type == "credit_card":
            print(f"Processing credit card payment of {amount}")
            # Credit card-specific logic
        elif payment_type == "paypal":
            print(f"Processing PayPal payment of {amount}")
            # PayPal-specific logic
        elif payment_type == "bank_transfer":
            print(f"Processing bank transfer payment of {amount}")
            # Bank transfer-specific logic
        else:
            print("Unsupported payment type")
```

Conditional Complexity
Solution

Use **polymorphism**.

- Separate the common logic from the special case logic.
- Create a supertype (abstract class or interface) as placeholder for special case logic.
- Create subtypes to hold logic for each special case.

Patterns: Strategy, State

ACTIVITY

- **Task:** Groups will create a short explanation for each identified code smells, including:
- **What:** Name the code smells.
- **Where:** Point out the specific lines or sections where it occurs.
- **Why:** Briefly explain why it qualifies as that code smell and the potential issues it could cause (e.g., hard to maintain, error-prone).