Code Smells and Refactor

Why Refactor code?

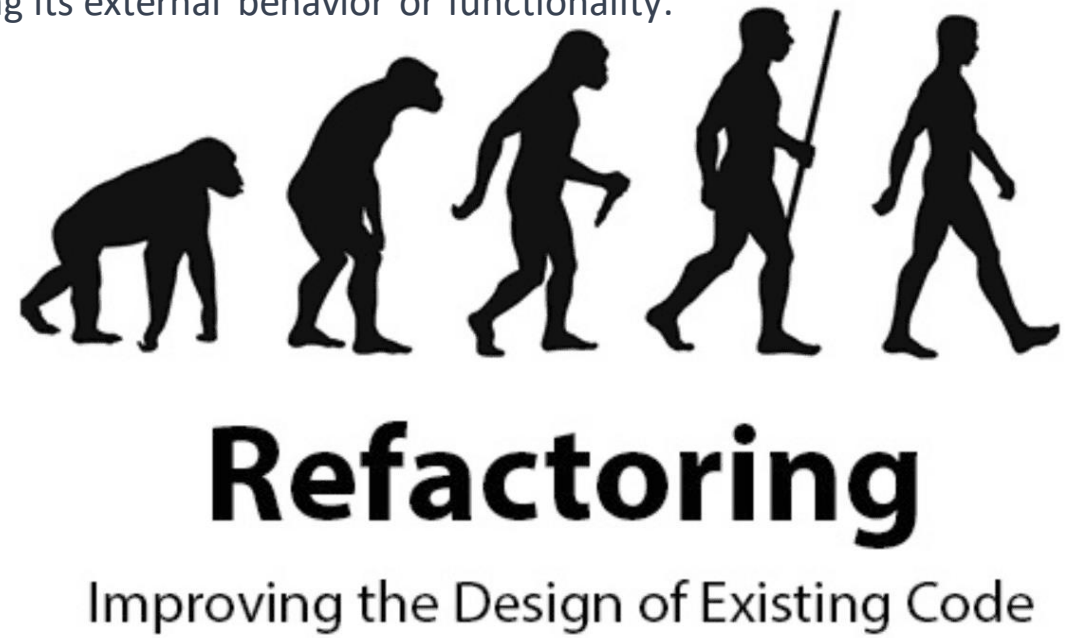
- Code smells
- Codes are evolving

What is code smells

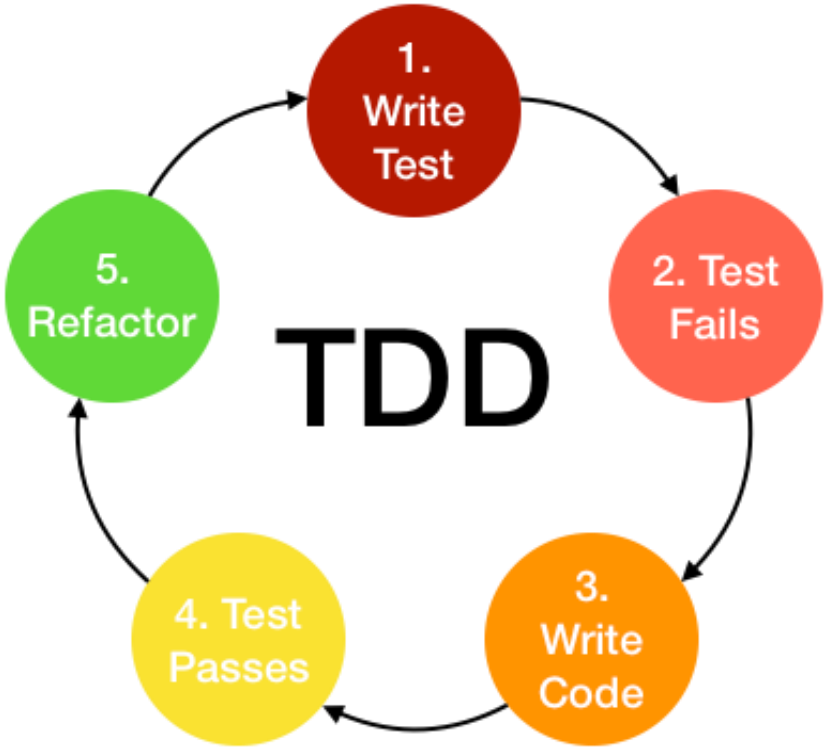
- Code smell refers to any symptom in the source code of a program that possibly indicates a deeper problem. It is a term used to describe code that does not adhere to best practices and could be a sign that the code needs to be refactored. Code smells don't stop the program from functioning, but they can make the code harder to maintain and understand.

What is refactoring

- Refactoring is the process of restructuring and improving the internal code of a software system without changing its external behavior or functionality.



When to Refactor code?



What benefits of Refactoring code?

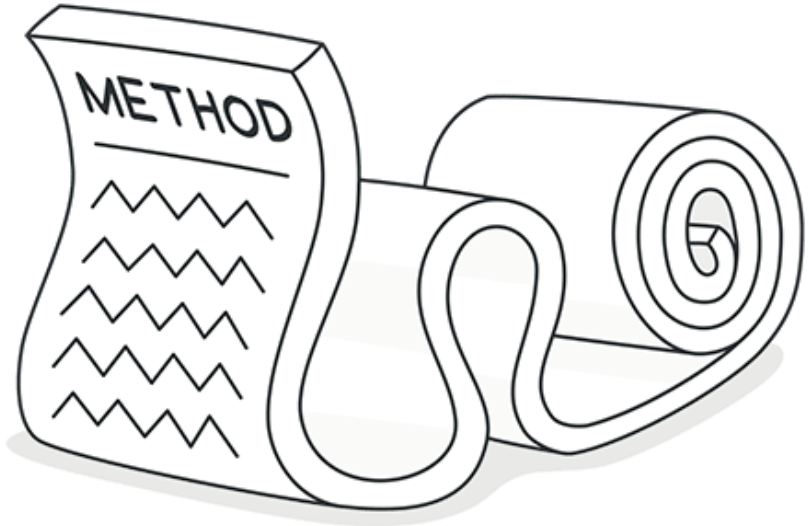
- Improving readability and maintainability
- Reducing complexity
- Improving performance
- Making it easier to extend
- Removing duplicated code
- Improving testability
- Adapting to changing requirements

Statistic shows:

- Rename variable: 600,776 commits
- Rename method: 157,815 commits
- Rename class: 99,264 commits
- Move method: 82,009 commits
- Move class: 65,364 commits
- Extract method: 50,401 commits
- Inline method: 39,309 commits
- Extract class: 9,009 commits
- Extract interface: 7,503 commits
- Extract superclass: 1,270 commits
- Pull up method: 1,388 commits
- Push down method: 100 commits

Bloater - Long Method

A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.



Bloater - Long Method, how to fix

- **Extract Method**: This is one of the most common refactoring techniques for addressing Long Method. You take a code fragment that can be grouped together, move it into a separate new method, and replace the old code with a call to the new method.
- **Replace Temp with Query**: If you have temporary variables that are taking up a lot of space, consider replacing them with a query method. This may also make it easier to use Extract Method afterward.
- **Introduce Parameter Object**: If the method has a long list of parameters, or you're passing related data, you can consolidate these parameters into a single object.
- **Preserve Whole Object**: Instead of passing several parts of an object to a method, pass the whole object.
- **Replace Method with Method Object**: If local variables make it difficult to apply Extract Method, turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.
- **Decompose Conditional**: If you have a complex conditional (if-else statements), this can be a sign that the method is handling too many cases and it can be beneficial to extract methods for the condition, then-clause, and else-clause.

Bloater - Long Method (Demo: bad)

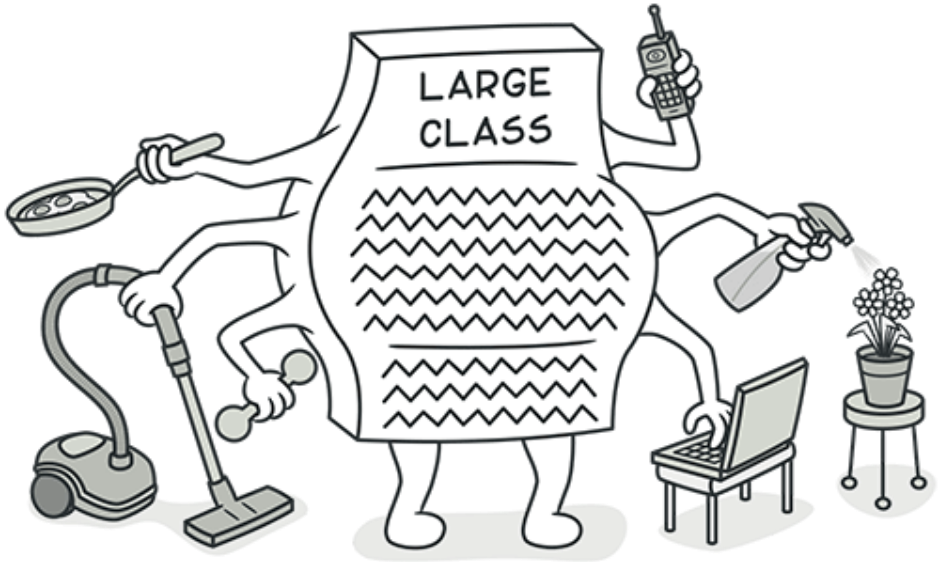
```java
public class ReportGenerator {

    public void generateReport(List<String> data, String title, String header) {
        // Print title
        System.out.println(title);
        System.out.println("====================");

        // Print header
        System.out.println(header);

        // Print data
        for (String item : data) {
            System.out.println(item);
        }

        // Calculate and print summary
        int totalItems = data.size();
        System.out.println("Total items: " + totalItems);

        // Adding some fake processing to make the method longer
        // ...
    }
}
```

Bloater - Long Method (Demo: Good)

```java
public class ReportGenerator {

    public void generateReport(List<String> data, String title, String header) {
        printTitle(title);
        printHeader(header);
        printData(data);
        printSummary(data);
    }

    private void printTitle(String title) {
        System.out.println(title);
        System.out.println("====================");
    }

    private void printHeader(String header) {
        System.out.println(header);
    }

    private void printData(List<String> data) {
        for (String item : data) {
            System.out.println(item);
        }
    }

    private void printSummary(List<String> data) {
        int totalItems = data.size();
        System.out.println("Total items: " + totalItems);
    }
}
```

Bloater – Large Class

A class should have a single responsibility, and when it becomes too large, it's usually doing too much.



Bloater – Large Class (Fixes)

- **Extract Class:** if part of the behavior of the large class can be spun off into a separate component.
- **Extract Subclass:** if part of the behavior of the large class can be implemented in different ways or is used in rare cases.
- **Extract Interface:** if it's necessary to have a list of the operations and behaviors that the client can use.

Bloater – Large Class (Demo : Bad)

```java
public class Car {

    private String make;
    private String model;
    private int year;
    private double fuelLevel;

    // Constructor and basic properties...

    // Behavior typical for a car
    public void accelerate() {
        // logic to accelerate
    }

    public void brake() {
        // logic to brake
    }

    public void refuel(double amount) {
        // logic to refuel
    }

    // Logic for navigation - not really the responsibility of the Car class
    public void calculateRoute(String destination) {
        // logic to calculate route
    }

    public void startNavigation() {
        // logic to start navigation
    }

    // Logic for playing music - not really the responsibility of the Car class
    public void playSong(String songName) {
        // logic to play a song
    }

    public void stopMusic() {
        // logic to stop the music
    }
```

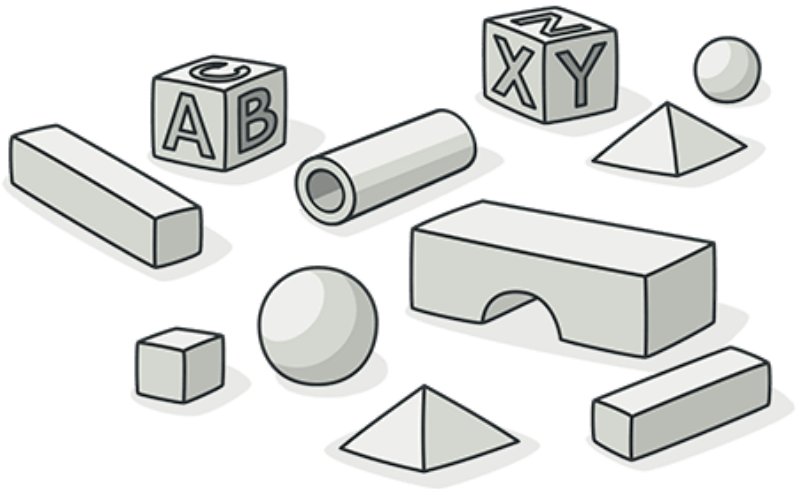
Bloater - Long Method (Demo: Good)

```java
public class Car {

    private String make;
    private String model;
    private int year;
    private double fuelLevel;

    // Constructor and basic properties...

    public void accelerate() {
        // logic to accelerate
    }

    public void brake() {
        // logic to brake
    }

    public void refuel(double amount) {
        // logic to refuel
    }
}
public class NavigationSystem {

    public void calculateRoute(String destination) {
        // logic to calculate route
    }

    public void startNavigation() {
        // logic to start navigation
    }
}
```

```java
public class MusicPlayer {

    public void playSong(String songName) {
        // logic to play a song
    }

    public void stopMusic() {
        // logic to stop the music
    }
}
public class MaintenanceManager {

    public void scheduleMaintenance(String date) {
        // logic to schedule maintenance
    }

    public void contactMechanic(String mechanicName) {
        // logic to contact a mechanic
    }
}
```

Bloater – Primitive Obsession

Using primitive data types to represent domain ideas. For example, using strings to represent currencies or phone numbers instead of creating custom classes.



Bloater – Primitive (Fixes)

- **Replace Data Value with Object**: If you are using primitive data types to represent domain entities, you can create a class to encapsulate these primitives and their associated behavior.
- **Replace Type Code with Class/Subclasses/State-Strategy**: If you are using codes (such as integers or strings) to represent different types of things, you can replace these codes with classes, subclasses, or use the state or strategy patterns.
- **Introduce Parameter Object or Preserve Whole Object**: If methods take multiple primitive values as parameters which can be logically grouped, you can pass an object instead.
- **Replace Array with Object**: If you are using arrays to group related data, you can create a class to encapsulate the elements of the array and their associated behavior.
- **Extract Class**: If a class has too many primitive fields, consider extracting some of them into a new class that represents a more focused concept.

Bloater - Primitive (Demo: Bad)

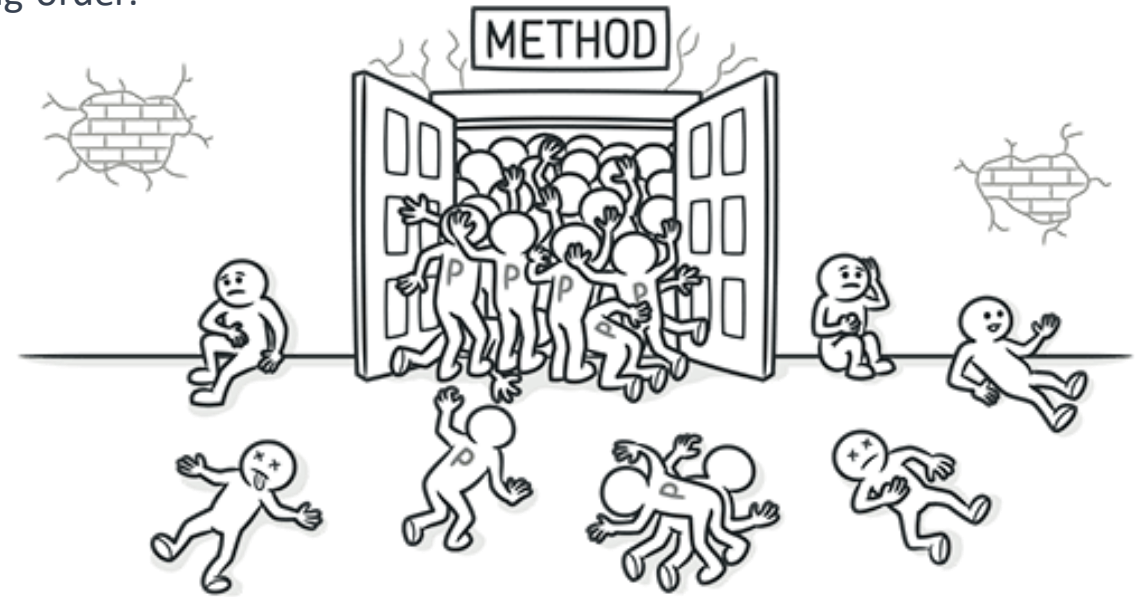
```java
public class User {
    private String name;
    private String street;
    private String city;
    private String postalCode;

    public User(String name, String street, String city, String postalCode) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.postalCode = postalCode;
    }

    public void printAddress() {
        System.out.println(street + ", " + city + ", " + postalCode);
    }
}
```

Bloater - Primitive (Demo : Good)

```java
public class Address {
    private String street;
    private String city;
    private String postalCode;

    public Address(String street, String city, String postalCode) {
        this.street = street;
        this.city = city;
        this.postalCode = postalCode;
    }

    @Override
    public String toString() {
        return street + ", " + city + ", " + postalCode;
    }
}
public class User {
    private String name;
    private Address address;

    public User(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public void printAddress() {
        System.out.println(address);
    }
}
```

Bloater – Long Parameter List

When a method has too many parameters. Having too many parameters can make a method difficult to understand and maintain. It also increases the chances of errors as callers may pass arguments in the wrong order.



Bloater – Long Parameter List (Fixes)

- **Introduce Parameter Object**: Taking a group of parameters that naturally fit together (usually because they are closely related) and replacing them with an object that encapsulates them.
- **Preserve Whole Object**: Sometimes, you may find that you are passing in several attributes from the same object as parameters. In such cases, it makes sense to just pass the whole object instead.
- **Replace Parameter with Method Call**: If you can obtain the data in one parameter by making a call to a method, then you can remove the parameter and let the method retrieve the data itself.
- **Replace Parameter with Query**: Similar to replacing a parameter with a method call, this involves replacing a parameter with a method that can retrieve or calculate the needed data.

Bloater - Long Parameter List (Demo: bad)

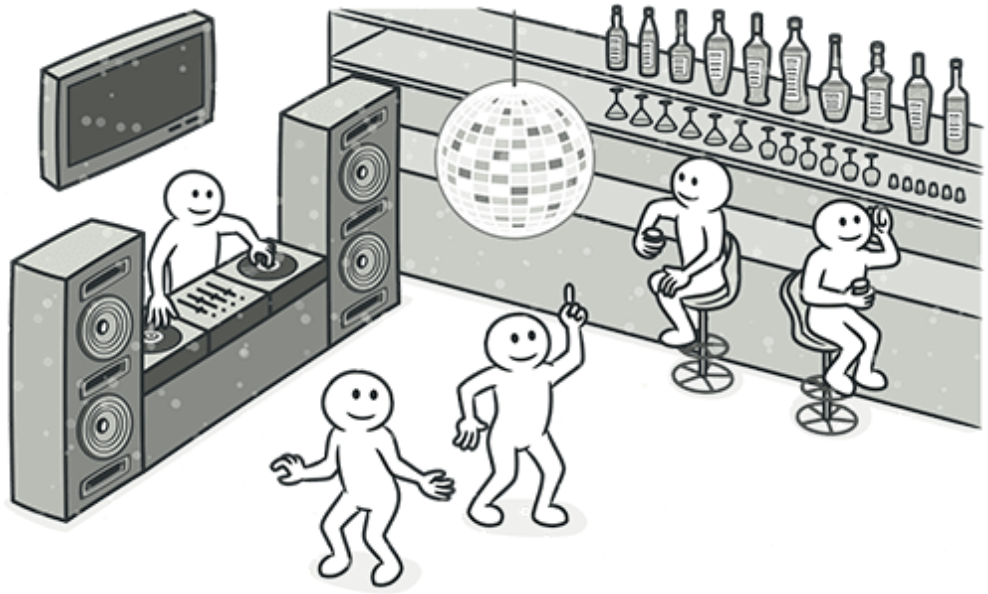
```java
public class MortgageCalculator {

    public double calculateMonthlyPayment(
            double principal,
            double annualInterestRate,
            int loanPeriodInYears,
            double propertyTaxRate,
            double insuranceRate,
            double homeownersAssociationFees) {

        double monthlyInterestRate = annualInterestRate / 12 / 100;
        int numberOfPayments = loanPeriodInYears * 12;
        double monthlyPayment = (principal * monthlyInterestRate * Math.pow(1 + monthlyInterestRate, numberOfPayments))
                / (Math.pow(1 + monthlyInterestRate, numberOfPayments) - 1);

        double monthlyPropertyTax = (principal * propertyTaxRate) / 12;
        double monthlyInsurance = (principal * insuranceRate) / 12;

        return monthlyPayment + monthlyPropertyTax + monthlyInsurance + homeownersAssociationFees;
    }
}
```

Bloater - Long Parameter List (Demo: Good)

```java
public class MortgageParameters {
    public final double principal;
    public final double annualInterestRate;
    public final int loanPeriodInYears;
    public final double propertyTaxRate;
    public final double insuranceRate;
    public final double homeownersAssociationFees;

    public MortgageParameters(double principal, double annualInterestRate, int loanPeriodInYears,
                              double propertyTaxRate, double insuranceRate, double homeownersAssociationFees) {
        this.principal = principal;
        this.annualInterestRate = annualInterestRate;
        this.loanPeriodInYears = loanPeriodInYears;
        this.propertyTaxRate = propertyTaxRate;
        this.insuranceRate = insuranceRate;
        this.homeownersAssociationFees = homeownersAssociationFees;
    }
}
public class MortgageCalculator {

    public double calculateMonthlyPayment(MortgageParameters params) {

        double monthlyInterestRate = params.annualInterestRate / 12 / 100;
        int numberOfPayments = params.loanPeriodInYears * 12;
        double monthlyPayment = (params.principal * monthlyInterestRate * Math.pow(1 + monthlyInterestRate, numberOfPayments))
                / (Math.pow(1 + monthlyInterestRate, numberOfPayments) - 1);

        double monthlyPropertyTax = (params.principal * params.propertyTaxRate) / 12;
        double monthlyInsurance = (params.principal * params.insuranceRate) / 12;

        return monthlyPayment + monthlyPropertyTax + monthlyInsurance + params.homeownersAssociationFees;
    }
}
```

Bloater – Data Clumps

Data Clumps occur when you have a set of data items that are always used together.



Bloater – Data Clumps (Fixes)

- **Replace Data Value with Object**: Encapsulate related fields into a new class.
- **Extract Class**: Move clumped fields from an existing class into a new class.
- **Introduce Parameter Object**: Replace method parameters that are always passed together with an object.
- **Preserve Whole Object**: Pass the entire object instead of its individual attributes.

Bloater – Data Clumps (Demo: bad)

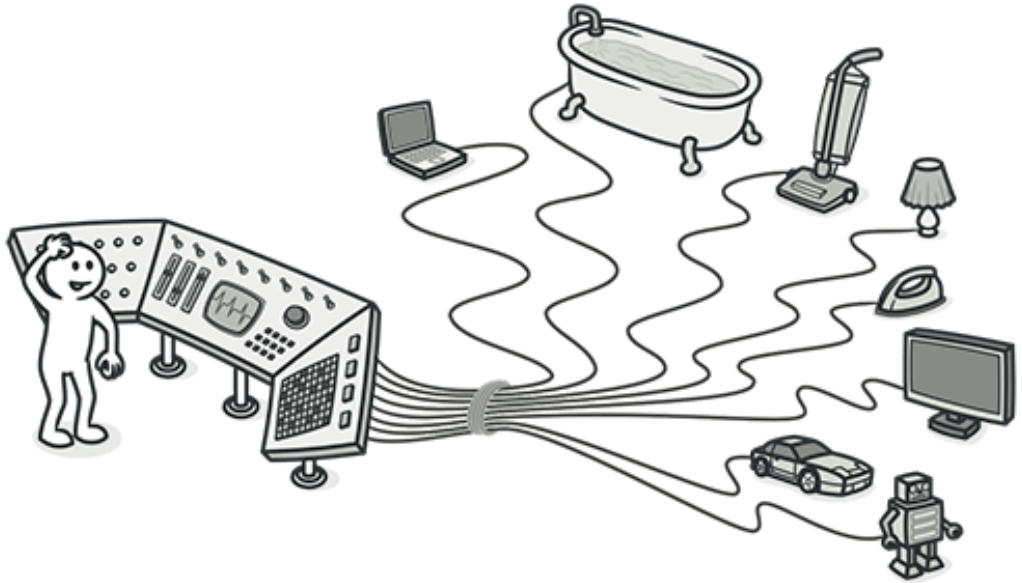
```java
public class User {
    private String name;
    private String street;
    private String city;
    private String postalCode;

    public User(String name, String street, String city, String postalCode) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.postalCode = postalCode;
    }

    // ... other methods ...
}
```

Bloater – Data Clumps (Demo: Good)

```java
public class User {
    private String name;
    private Address address;

    public User(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    // ... other methods ...
}

class Address {
    private String street;
    private String city;
    private String postalCode;

    public Address(String street, String city, String postalCode) {
        this.street = street;
        this.city = city;
        this.postalCode = postalCode;
    }

    // ... other methods ...
}
```

OO Abuser – Switch Statements

Duplicating the same **switch** structure in various places in the code, or it tends to change often for different reasons.



OO Abuser – Switch Statements (Fixes)

- **Extract Method:** To isolate switch and put it in the right class, you may need **Extract Method** and then **Move Method**.
- **Replace Conditional with Polymorphism**: Create subclasses matching the branches of the **switch** statement, and override a method in these subclasses to replace the **switch**.
- **Replace Type Code with Subclasses**: If the **switch** is based on type codes, create subclasses for each type code, and use them instead.
- **Replace Type Code with State/Strategy**: Similar to replacing with subclasses but use state or strategy pattern if runtime change is needed.
- **Replace Parameter with Explicit Methods**: If the **switch** is just selecting different behavior based on an identifier or type code, consider using separate explicit methods instead.

OO Abuser – Switch Statements (Demo: bad)

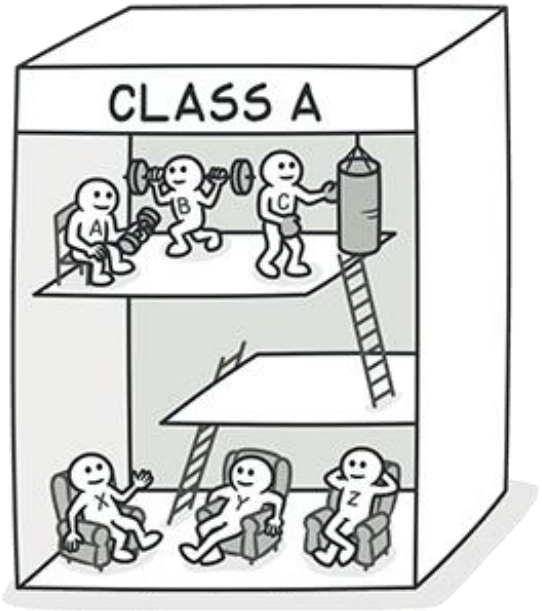
```
1  public class Animal {
2      private String type;
3
4      public Animal(String type) {
5          this.type = type;
6      }
7
8      public String makeSound() {
9          switch (type) {
10             case "dog":
11                 return "bark";
12             case "cat":
13                 return "meow";
14             case "cow":
15                 return "moo";
16             default:
17                 return "unknown";
18         }
19     }
20 }
```

OO Abuser – Switch Statements (Demo: Good)

```java
public interface Animal {
    String makeSound();
}
public class Cat implements Animal {
    @Override
    public String makeSound() {
        return "meow";
    }
}
public class Cow implements Animal {
    @Override
    public String makeSound() {
        return "moo";
    }
}
public class Dog implements Animal {
    @Override
    public String makeSound() {
        return "bark";
    }
}
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
        Animal cow = new Cow();

        System.out.println("Dog: " + dog.makeSound());
        System.out.println("Cat: " + cat.makeSound());
        System.out.println("Cow: " + cow.makeSound());
    }
}
```

OO Abuser – Temporary Field

- when an object has a field or variable that gets used only occasionally, or under certain circumstances. For the rest of the time, that field is either idle or simply carrying null or some default value.



OO Abuser – Temporary Field (Fixes)

- **Extract Class:** You can create a new class, move the relevant methods and fields into that new class, and then reference it from the old class.
- **Replace Method with Method Object:** You can turn the method into its own object, so the local variables become fields on that object. You can then decompose the method into other methods on the same object.

OO Abuser – Temporary Field (Demo: bad)

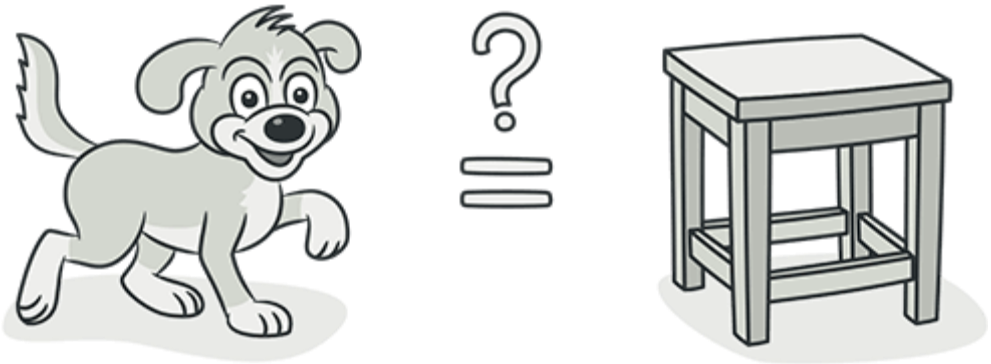
```java
public class User {
    private String name;
    private String email;
    private String tempPassword;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void resetPassword() {
        this.tempPassword = generateTempPassword();
        sendPasswordToEmail();
        this.tempPassword = null;
    }

    private String generateTempPassword() {
        // generates a temporary password
    }

    private void sendPasswordToEmail() {
        // sends the password to user's email
        // uses tempPassword
    }
}
```

OO Abuser – Temporary Field (Demo: good)

```java
public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void resetPassword() {
        PasswordReset passwordReset = new PasswordReset();
        passwordReset.resetPassword(this.email);
    }
}

public class PasswordReset {
    private String tempPassword;

    public void resetPassword(String email) {
        this.tempPassword = generateTempPassword();
        sendPasswordToEmail(email);
    }

    private String generateTempPassword() {
        // generates a temporary password
    }

    private void sendPasswordToEmail(String email) {
        // sends the password to user's email
        // uses tempPassword
    }
}
```

OO Abuser – Refused Bequest

- when a subclass only uses a small fraction of the properties and methods of its superclass, or does not use them at all, essentially refusing the inheritance ("bequest") from its parent. This may imply that the subclass and superclass are not conceptually similar enough to warrant an inheritance relationship.



OO Abuser – Refused Bequest （fix）

- **Replace Inheritance with Delegation:** The subclass can have a private field that references an instance of the superclass, and methods in the subclass can call through to methods in the superclass when appropriate.
- **Extract Subclass/Superclass or Interface:** If the subclass is using a small but coherent subset of the superclass's functionality, consider extracting that functionality into its own class or interface.

OO Abuser – Refused Bequest (Demo: bad)

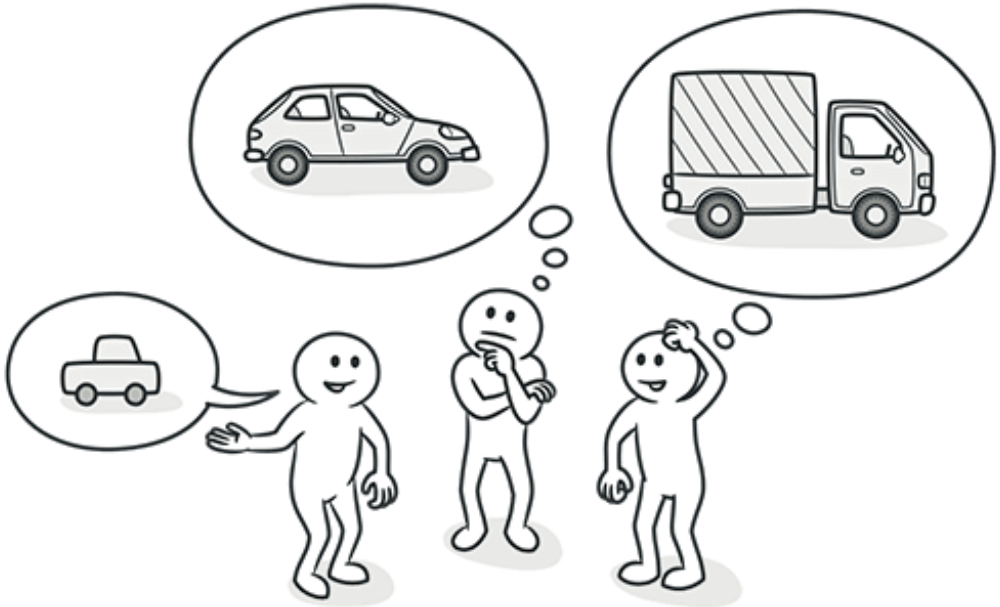
```
1  public class Bird {
2      public void fly() {}
3      public void eat() {}
4  }
5
6  public class Penguin extends Bird {
7      @Override
8      public void fly() {
9          throw new UnsupportedOperationException();
10     }
11 }
```

OO Abuser – Refused Bequest (Demo: good)

```
1  public interface Animal {
2      void eat();
3  }
4
5  public class FlyingBird implements Animal {
6      public void fly() {}
7      public void eat() {}
8  }
9
10 public class Penguin implements Animal {
11     public void eat() {}
12 }
```

OO Abuser – Alternative Classes with Different Interfaces

- When there are several classes that perform similar tasks but have different method names.



OO Abuser – Alternative Classes with Different Interfaces(fix）

- **Rename Method:** Make the method names consistent across similar classes.
- **Extract Interface:** Create an interface that declares common methods, and have the classes implement this interface.
- **Extract Superclass:** Move shared methods or fields to a common superclass that the original classes inherit from.
- **Change Method Signature:** Ensure similar methods across classes have matching signatures.

OO Abuser – A.C.D.I. (Demo: bad)

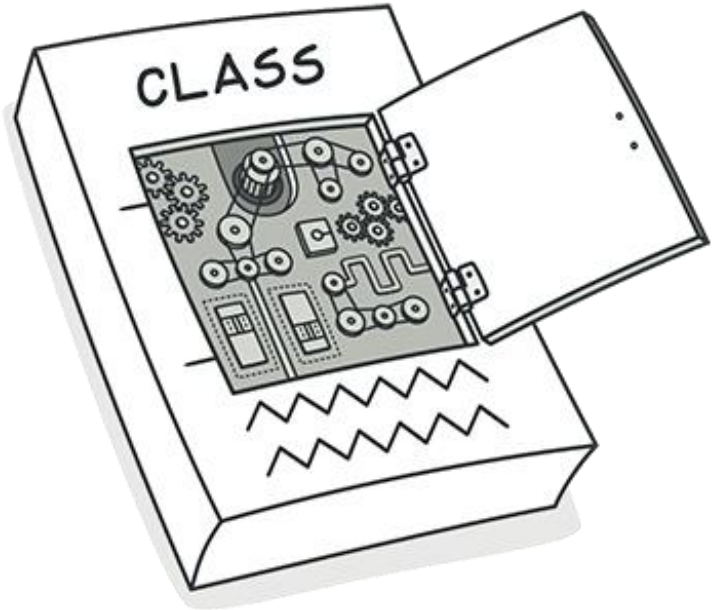
```java
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double calculateArea() {
        return Math.PI * Math.pow(radius, 2);
    }
}

public class Rectangle {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }
}
```

OO Abuser –
A.C.D.I. (Demo: good)

```java
public interface Shape {
    double getArea();
}

public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * Math.pow(radius, 2);
    }
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
}
```

Change Preventers – Divergent Change

- When make changes to a class each time there are changes to different parts of the application, that class may be responsible for too many different things.



Change Preventers – Divergent Change(fix）

- **Extract Class Refactoring**: identify methods and fields in the class that can be isolated into a separate class. This involves creating a new class and moving the relevant methods and fields from the old class into the new class.

Change Preventers – Divergent Change(Demo: bad）

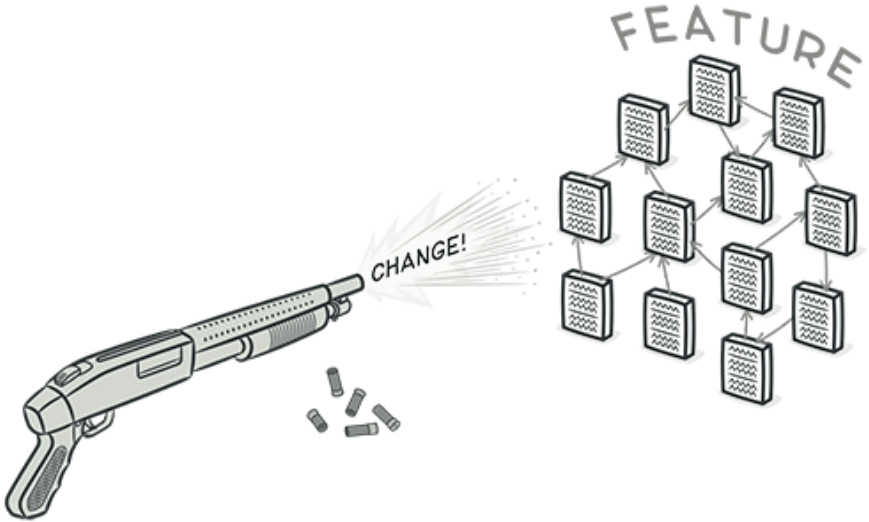
```java
public class Book {
    private String title;
    private String author;
    private double price;
    private int stockCount;

    public Book(String title, String author, double price, int stockCount) {
        this.title = title;
        this.author = author;
        this.price = price;
        this.stockCount = stockCount;
    }

    // ... getters and setters ...

    public void displayDetails() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Price: " + price);
    }

    public void updateStockCount(int newCount) {
        this.stockCount = newCount;
        System.out.println("Updated stock count: " + stockCount);
    }
}
```

Change Preventers – Divergent Change(Demo: good）

```java
public class Book {
    private String title;
    private String author;
    private double price;
    private int stockCount;

    public Book(String title, String author, double price, int stockCount) {
        this.title = title;
        this.author = author;
        this.price = price;
        this.stockCount = stockCount;
    }

    // ... getters and setters ...

    public void updateStockCount(int newCount) {
        this.stockCount = newCount;
    }
}

public class BookDisplay {
    public void displayDetails(Book book) {
        System.out.println("Title: " + book.getTitle());
        System.out.println("Author: " + book.getAuthor());
        System.out.println("Price: " + book.getPrice());
    }
}
```

Change Preventers – Shotgun Surgery

- "Shotgun Surgery" is a code smell which refers to a situation where a single change in the codebase requires changes in multiple classes or methods simultaneously. This kind of code smell indicates that the responsibilities of the changed class or method are too spread out across the system, leading to high coupling.



Change Preventers – Shotgun Surgery (fix)

- **Move Method** or **Move Field**: If a method or a field is more frequently used in another class than the one where it is currently located, it may be a good idea to move it to that class. This will bring related pieces of code together and make it easier to maintain the codebase.
- **Inline Class**: If a class isn't doing enough to earn your attention and seems more like a satellite to another class, it might be beneficial to merge it with another class.
- **Extract Class**: If you have a class doing work that should be done by two, you can create a new class and move the relevant methods and fields from the old class into the new class.

Change Preventers – Shotgun Surgery (Demo: bad)

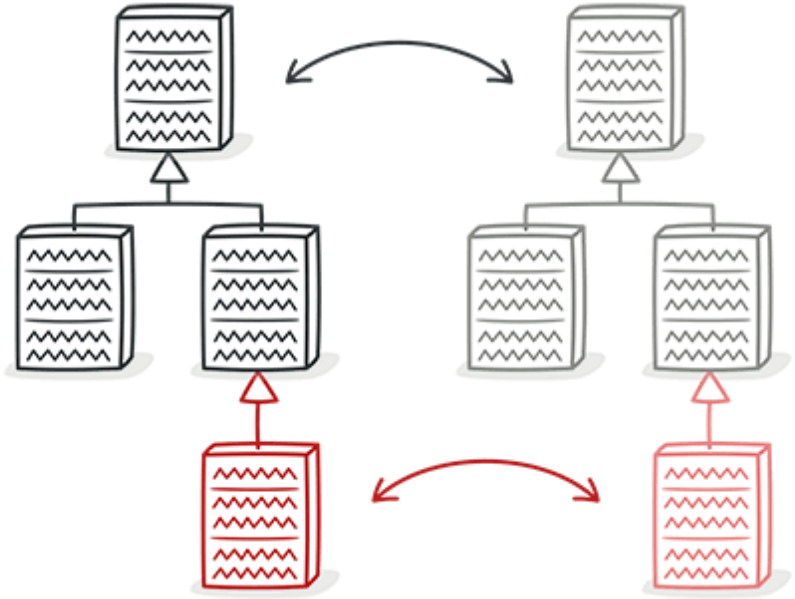
```java
public class User {
    private String username;
    private String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    // ... getters and setters ...
}

public class UserDatabase {
    private Map<String, User> database;

    public UserDatabase() {
        this.database = new HashMap<>();
    }

    public void add(User user) {
        if (user.getUsername().length() < 5 || user.getPassword().length() < 8) {
            throw new IllegalArgumentException("Username must be at least 5 characters.
            Password must be at least 8 characters.");
        }
        if (database.containsKey(user.getUsername())) {
            throw new IllegalArgumentException("Username already exists.");
        }
        database.put(user.getUsername(), user);
    }

    // ... other database methods ...
}
```

Change Preventers – Shotgun Surgery (Demo: good)

```java
public class User {
    private String username;
    private String password;

    public User(String username, String password) {
        if (username.length() < 5 || password.length() < 8) {
            throw new IllegalArgumentException("Username must be at least 5 characters.
            Password must be at least 8 characters.");
        }
        this.username = username;
        this.password = password;
    }

    // ... getters and setters ...
}

public class UserDatabase {
    private Map<String, User> database;

    public UserDatabase() {
        this.database = new HashMap<>();
    }

    public void add(User user) {
        if (database.containsKey(user.getUsername())) {
            throw new IllegalArgumentException("Username already exists.");
        }
        database.put(user.getUsername(), user);
    }

    // ... other database methods ...
}
```

Change Preventers – Parallel Inheritance Hierarchies

- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.



Change Preventers – Parallel Inheritance Hierarchies (fix)

- **Move Method or Move Field**: Try moving methods or fields from one hierarchy to the other if they are more relevant there. This way, you're trying to reduce the dependency on the second hierarchy.
- **Use Delegation Over Inheritance**: Rather than creating subclasses in the second hierarchy, modify the original class in the second hierarchy to reference an object of the subclass in the first hierarchy. This way, the second hierarchy doesn't need to mimic the subclasses of the first hierarchy.

Change Preventers – Parallel Inheritance Hierarchies (Demo: bad)

```
1  // Parallel Hierarchy 1
2  public class Employee {
3      // ...
4  }
5
6  public class Engineer extends Employee {
7      // ...
8  }
9
10 public class Manager extends Employee {
11     // ...
12 }
13
14 // Parallel Hierarchy 2
15 public class EmployeeRecord {
16     // ...
17 }
18
19 public class EngineerRecord extends EmployeeRecord {
20     // ...
21 }
22
23 public class ManagerRecord extends EmployeeRecord {
24     // ...
25 }
```

Change Preventers – Parallel Inheritance Hierarchies (Demo: good)

```java
// Hierarchy 1
public class Employee {
    // ...
}

public class Engineer extends Employee {
    // ...
}

public class Manager extends Employee {
    // ...
}

// Hierarchy 2
public class EmployeeRecord {
    private Employee employee;

    // You can now delegate any Employee-specific behavior to the employee object
    public EmployeeRecord(Employee employee) {
        this.employee = employee;
    }

    // Example delegation method
    public void someEmployeeMethod() {
        this.employee.someEmployeeMethod();
    }
    // ...
}
```

Dispensable – unneeded code

Too much Comments

A method is filled with explanatory comments.

Duplicate Code

Two code fragments look almost identical.

Lazy Class

Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted.

Data Class

A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own.

Dead Code

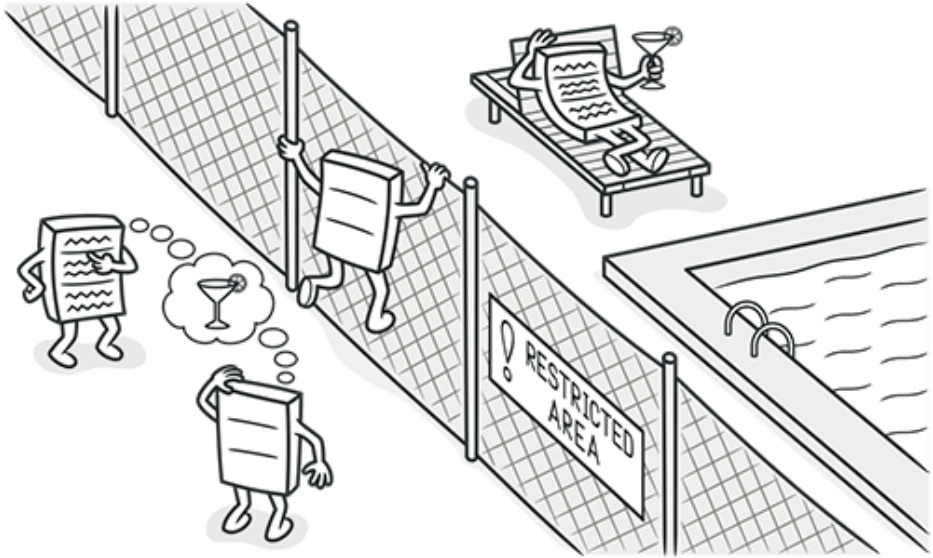
A variable, parameter, field, method or class is no longer used (usually because it's obsolete).

Speculative Generality

There's an unused class, method, field or parameter.

Couplers – Feature Envy

- A method accesses the data of another object more than its own data.



Couplers – Feature Envy (fix)

- **Move Method**: If a method uses features from another class more than its own, move the method to that other class.
- **Move Field**: If a field is used more in another class, move the field to that class.
- **Extract Class**: If a class shows envy towards many features of another class, create a new class encapsulating these fields and methods.

Couplers – Feature Envy (Demo: bad)

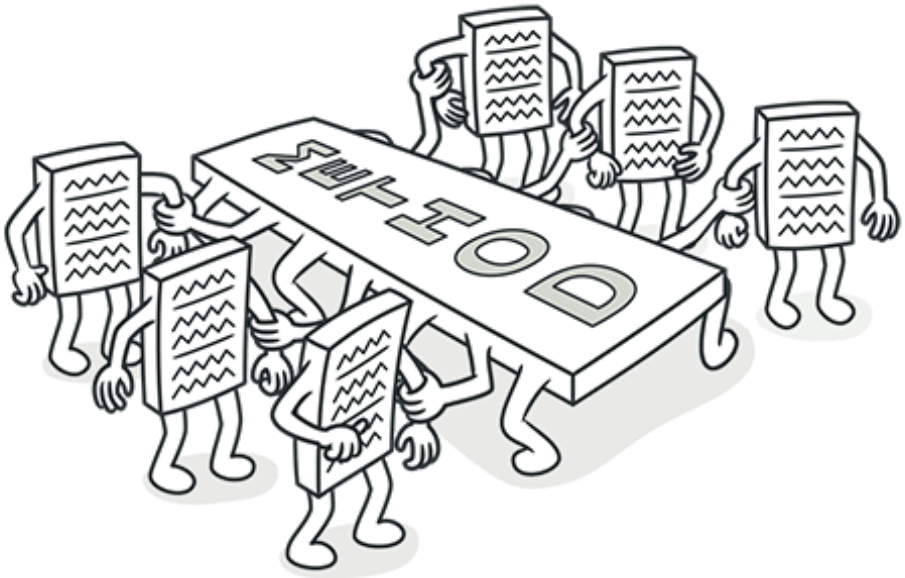
```java
// Initial code with feature envy
public class Customer {
    public String name;
    // other attributes...
}

public class Order {
    private Customer customer;
    private String customerName;

    public Order(Customer customer) {
        this.customer = customer;
        this.customerName = customer.name;
    }

    public String getCustomerName() {
        return this.customerName;
    }
}
```

Couplers – Feature Envy (Demo: good)

```java
// After refactoring
public class Customer {
    public String name;

    public String getName() {
        return this.name;
    }
    // other attributes...
}

public class Order {
    private Customer customer;

    public Order(Customer customer) {
        this.customer = customer;
    }

    public String getCustomerName() {
        return this.customer.getName();
    }
}
```

Couplers – Inappropriate Intimacy

- One class uses the internal fields and methods of another class.



Couplers – Inappropriate Intimacy (fix)

- **Move Method/Field**: If one class heavily uses another class's method or field, move it to the user class.
- **Change Association**: If two classes are overly dependent, make their relationship unidirectional.
- **Replace Inheritance/Extract Class**: If a subclass overuses a superclass's features, consider using delegation or creating a new class.
- **Hide Delegate/Encapsulate Field**: Hide excessive interactions between two classes by creating intermediary methods or make a public field private and use accessors.

Couplers – Inappropriate Intimacy (Demo: bad)

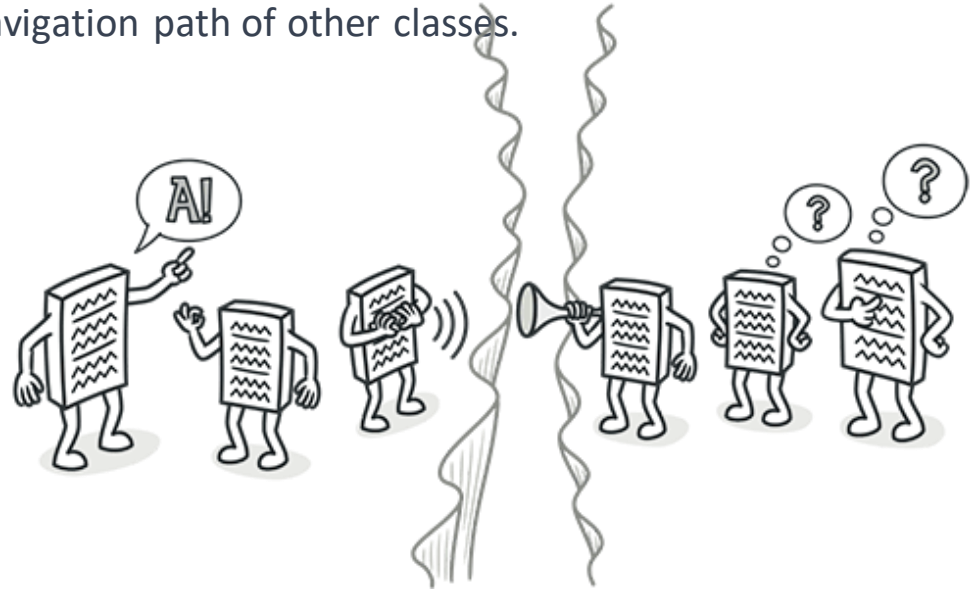
```java
public class Address {
    public String street;
    public String city;
    public String country;
    //...
}

public class Person {
    private Address address;

    public Person(Address address) {
        this.address = address;
    }

    public void relocate(String newStreet, String newCity, String newCountry) {
        address.street = newStreet;
        address.city = newCity;
        address.country = newCountry;
    }
    //...
}
```

Couplers – Inappropriate Intimacy (Demo: good)

```java
public class Address {
    private String street;
    private String city;
    private String country;

    public Address(String street, String city, String country) {
        this.street = street;
        this.city = city;
        this.country = country;
    }

    public void updateAddress(String street, String city, String country) {
        this.street = street;
        this.city = city;
        this.country = country;
    }
    //...
}

public class Person {
    private Address address;

    public Person(Address address) {
        this.address = address;
    }

    public void relocate(String newStreet, String newCity, String newCountry) {
        address.updateAddress(newStreet, newCity, newCountry);
    }
    //...
}
```

Couplers – Message Chains

when a client requests another object, that object requests yet another one, and so on. These chains can be seen as a sequence of method calls or requests for objects, creating a dependency on the internal structure or navigation path of other classes.



Couplers – Message Chains (fix)

- **Hide Delegate**: Create a method in the server class that hides the delegation and keeps the client class from knowing about the chained classes.
- **Extract Method**: Create a new method in the client class that makes the call to the server class, and replace the chained call with a call to the new method.
- **Middle Man**: If a class is only passing calls to other classes, consider removing it entirely.

Couplers – Message Chains (Demo: bad)

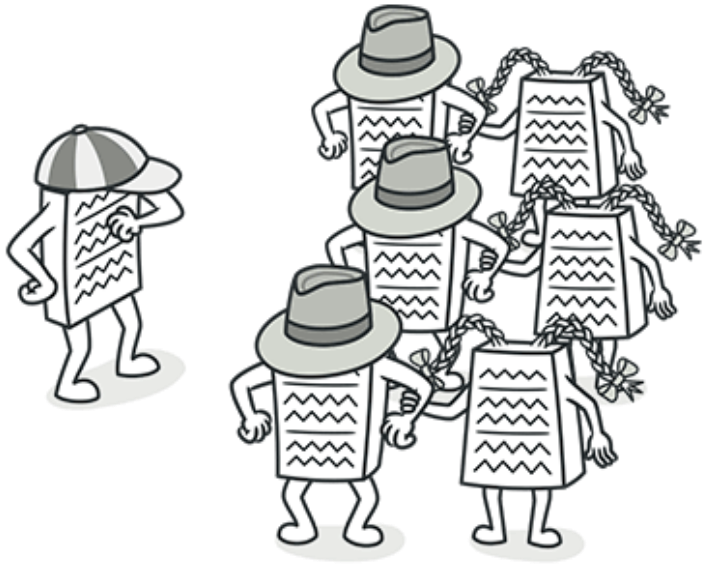
```java
class Product {
    private String name;

    public Product(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
class Customer {
    private Order order;

    public Customer(Order order) {
        this.order = order;
    }

    public Order getOrder() {
        return this.order;
    }
}
class Client {
    public static void main(String[] args) {
        Product product = new Product("Apple");
        Order order = new Order(product);
        Customer customer = new Customer(order);

        // Message Chain
        String productName = customer.getOrder().getProduct().getName();
        System.out.println(productName);
    }
}
class Client {
    public static void main(String[] args) {
        Product product = new Product("Apple");
        Order order = new Order(product);
        Customer customer = new Customer(order);

        // Message Chain
        String productName = customer.getOrder().getProduct().getName();
        System.out.println(productName);
    }
}
```

Couplers – Message Chains (Demo: good)

```java
class Product {
    private String name;

    public Product(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}

class Order {
    private Product product;

    public Order(Product product) {
        this.product = product;
    }

    public String getProductName() {
        return this.product.getName();
    }
}

class Customer {
    private Order order;

    public Customer(Order order) {
        this.order = order;
    }

    public String getProductName() {
        return this.order.getProductName();
    }
}
class Client {
    public static void main(String[] args) {
        Product product = new Product("Apple");
        Order order = new Order(product);
        Customer customer = new Customer(order);

        // Message Chain is removed
        String productName = customer.getProductName();
        System.out.println(productName);
    }
}
```

Couplers – Middle Man

A Middle Man is a class that seems to do a lot of delegating, but doesn't really do much work itself.



Couplers – Middle Man (Fix)

- Find classes that act as simple pass-throughs or are excessively delegating tasks to other classes.
- Replace all calls to the Middle Man class with direct calls to the class that the Middle Man is delegating to.
- Once all references to the M ed, if the class doesn't perform any other functions, conside

```
class RealPrinter {
    void print() {
        System.out.println("The printer is printing.");
    }
}

class Client {
    private RealPrinter realPrinter;

    void clientPrint() {
        realPrinter.print();
    }
}
```

Couplers – Middle Man (Demo : bad)

```
1  class RealPrinter {
2      void print() {
3          System.out.println("The printer is printing.");
4      }
5  }
6
7  class Printer {
8      private RealPrinter realPrinter;
9
10     void print() {
11         realPrinter.print();
12     }
13 }
```

Couplers – Middle Man (Demo : good)

```
1  class RealPrinter {
2      void print() {
3          System.out.println("The printer is printing.");
4      }
5  }
6
7  class Client {
8      private RealPrinter realPrinter;
9
10     void clientPrint() {
11         realPrinter.print();
12     }
13 }
```