

# Announcements

- Quiz 3 out all day tomorrow – Honorlock based open notes
- Code smells assignment due today. Work in class.
- Last lecture this Thursday.
- Project 2 Due soon. Please check the class website.

CS3300 Introduction to Software Engineering

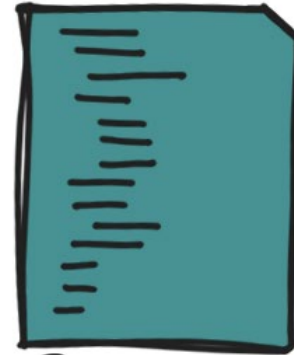
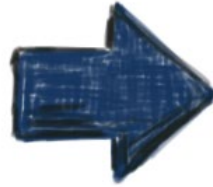
# Lecture 19: Code Smells

Dr. Nimisha Roy ▶ [nroy9@gatech.edu](mailto:nroy9@gatech.edu)

# Refactoring discussed last lecture..



Program



Refactored Program

Small, independent techniques to apply transformations to a program, with the goal of improving its design without changing its functionality

**Goal:** Keep program readable, understandable, and maintainable. Avoid small problems soon.

**Key Feature:** Behavior Preserving- make sure the program works after each step; typically small steps

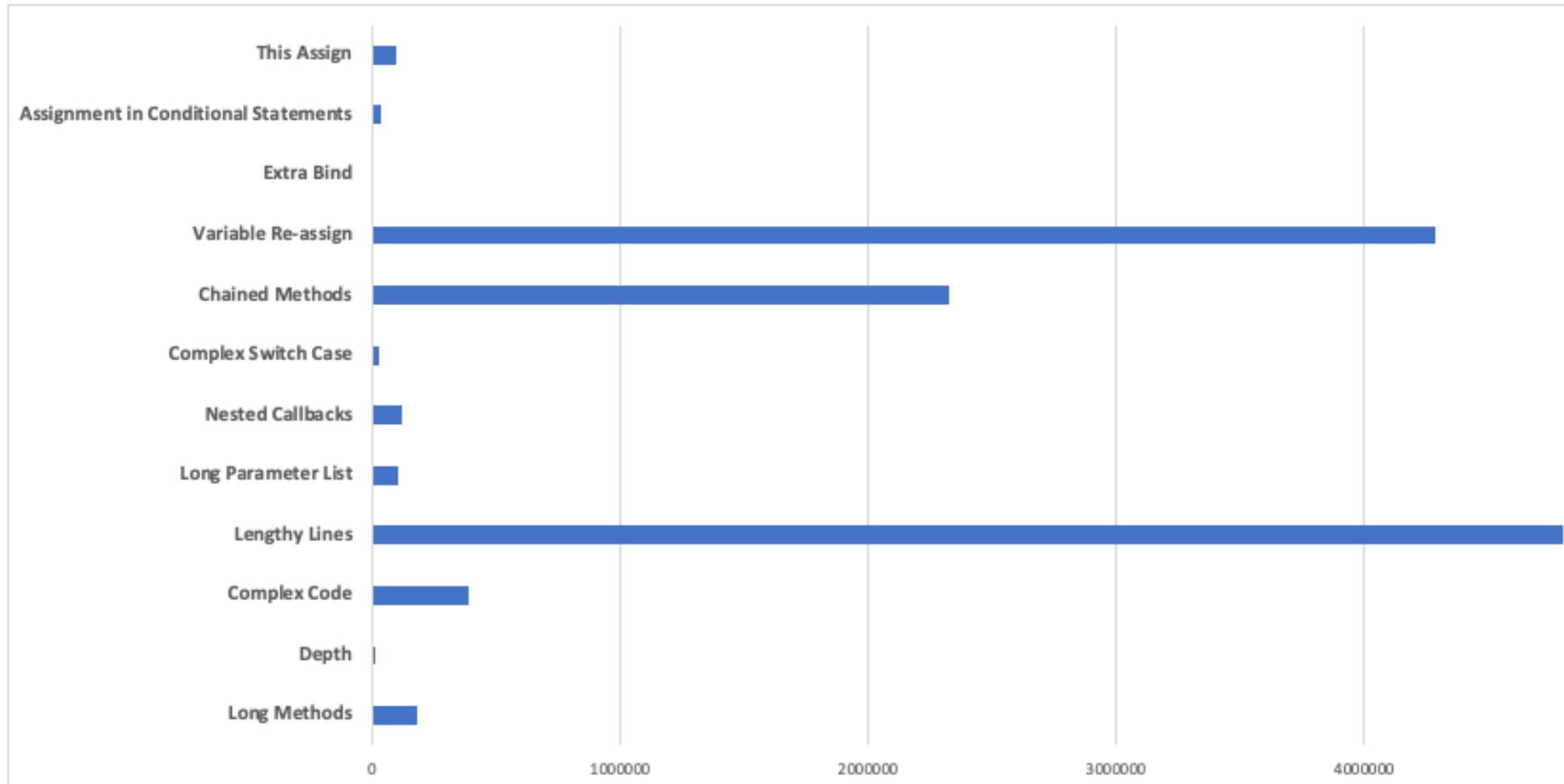
**How to know code needs Refactoring/Indicators of Problems in code :** Code Smells

# What is a Code Smell?

- Martin Fowler: *“a code smell is a surface indication that usually corresponds to a deeper problem in the system”*
  - Something that is quick to spot
  - Indicator of a bigger problem with your code
- Generally, you find code smells when examining the code, or doing refactoring

# Smells count across 15 JavaScript projects

“A large-scale empirical study of code smells in JavaScript projects” by D. Johannes, et al., *Software Quality Journal*, 3/22/2019

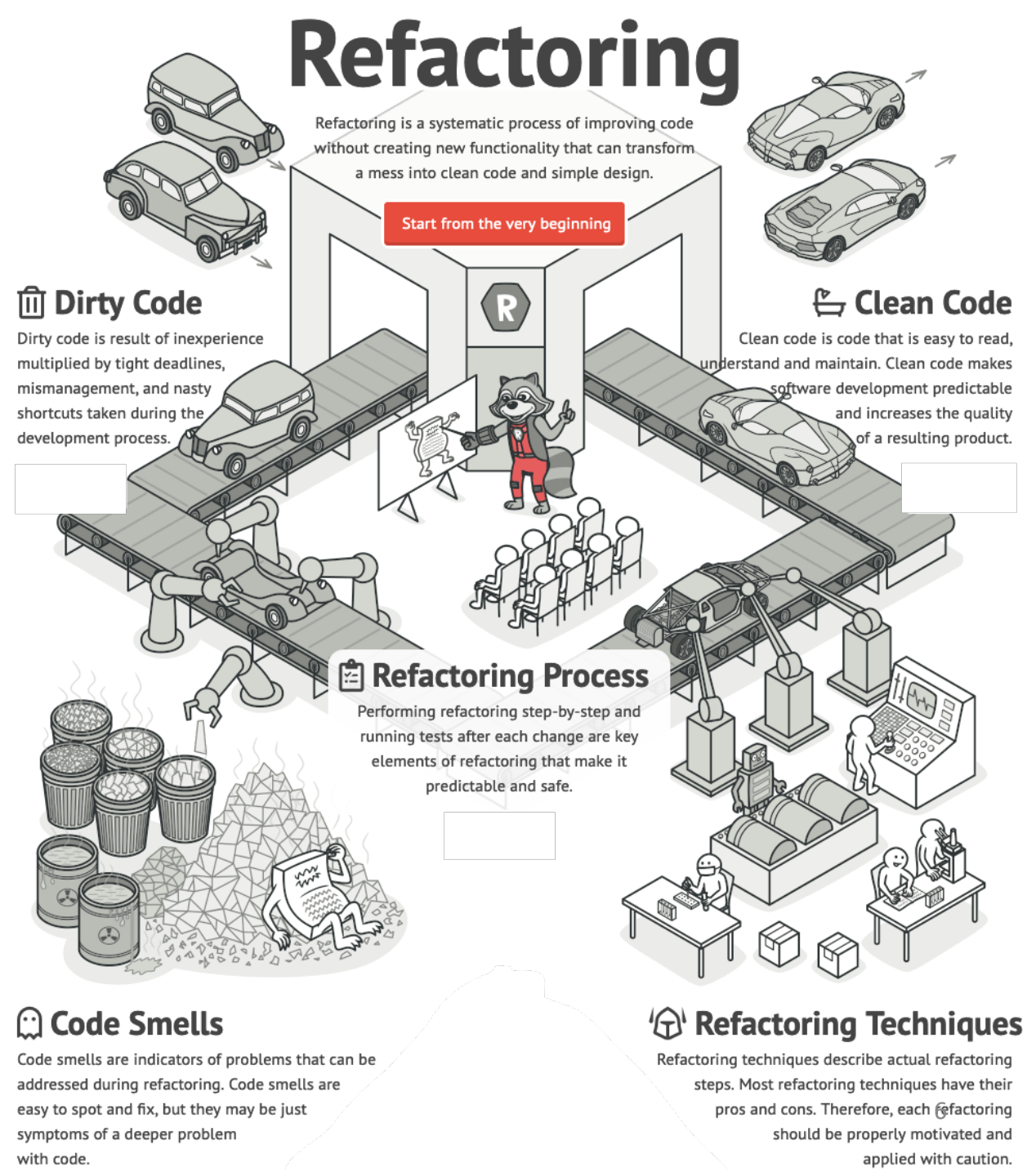


# Refactoring and Code Smells

Refactoring is a systematic process of **improving code** without creating new functionality that can transform a mess into **clean code** and **simple design**.

Code Smells help us find the problems

<https://refactoring.guru/refactoring>



# Code Smell: Summary

Group name	Short description	Smells in group	Treatment/Refactoring
<b>Bloaters</b>	something that has grown so large that it cannot be effectively handled.	<ul style="list-style-type: none"><li>-Long method</li><li>-Large class</li><li>-Long parameter list</li><li>-Data clumps</li></ul>	<ul style="list-style-type: none"><li>-Extract Method</li><li>-Extract class</li><li>-Replace parameters with method call/object</li><li>- Preserve whole object</li></ul>
<b>Object-Orientation Abusers</b>	cases where the solution does not fully exploit the possibilities of object-oriented design.	<ul style="list-style-type: none"><li>-Switch statements</li><li>-Refused Bequest (inheritance is being abused)</li></ul>	<ul style="list-style-type: none"><li>-Use polymorphism</li><li>-Push down method/-Eliminate inheritance</li></ul>
<b>Change Preventers</b>	smells that hinder changing or further developing the software.	<ul style="list-style-type: none"><li>-Divergent Change</li><li>-Shotgun surgery</li></ul>	<ul style="list-style-type: none"><li>-Extract Class, Method</li><li>-Move methods and fields</li></ul>
<b>Dispensables</b>	represent something unnecessary that should be removed from the source code.	<ul style="list-style-type: none"><li>-Lazy class</li><li>-Data class</li><li>-Duplicated code</li></ul>	<ul style="list-style-type: none"><li>-Inline class</li><li>-Move method/Extract Method/Encapsulation</li><li>-Extract Method</li></ul>
<b>Couplers</b>	a measure of how closely connected two routines or modules are	<ul style="list-style-type: none"><li>-Feature envy</li><li>-Inappropriate intimacy</li><li>-Middle man</li></ul>	<ul style="list-style-type: none"><li>-Move method/field</li><li>-Bidirectional to unidirectional association</li><li>-Inline Class</li></ul>

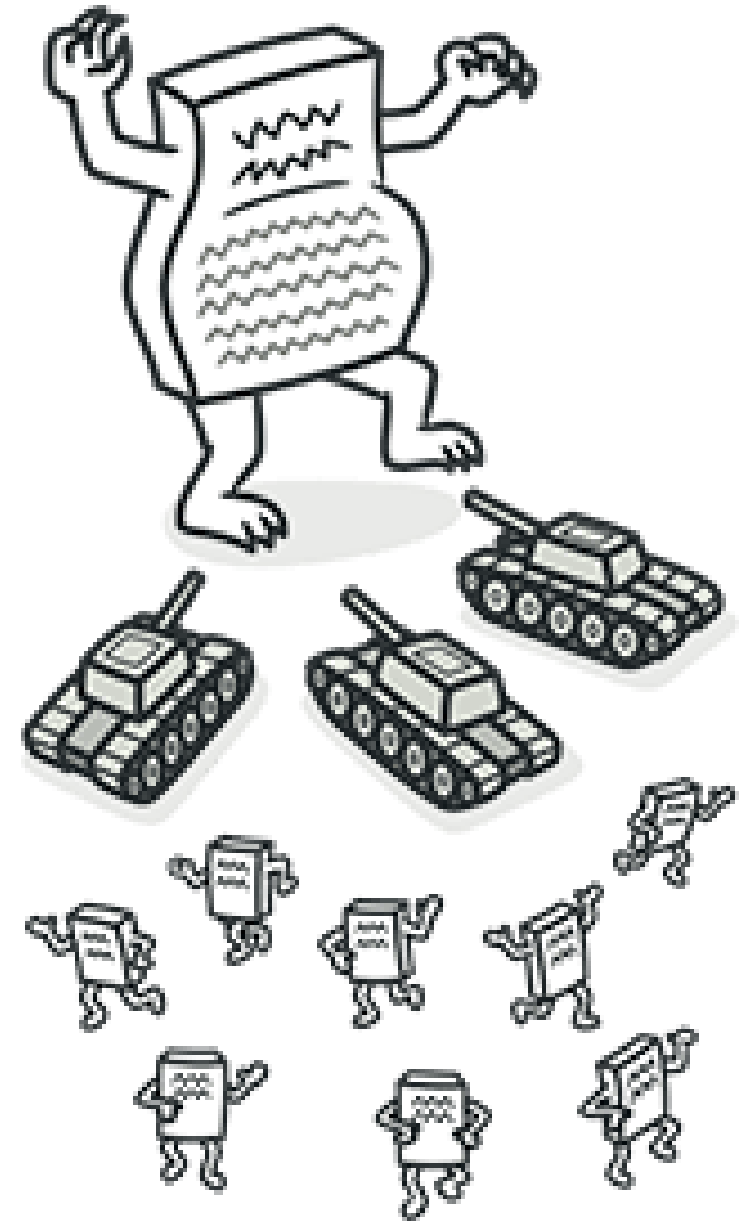


Work on the code smell  
activity

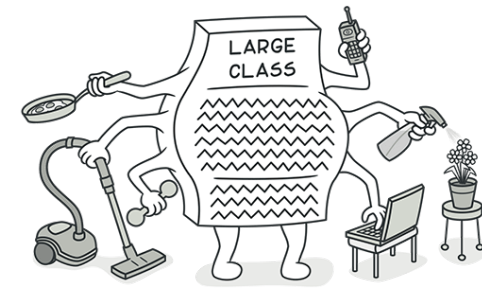


# Bloaters

Large Classes, Long Methods, Data Clumps, and Long Parameter List



# Bloaters: Large Class



## Signs/Symptoms

A class contains many fields/methods/lines of code. May have duplicated code.

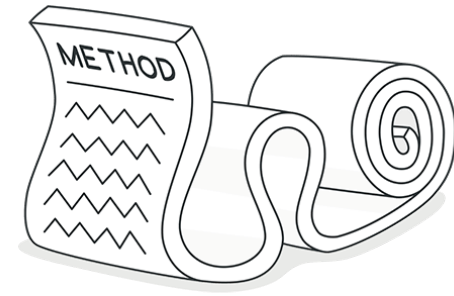
## Reasons

Developers may find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature.

## Treatment

Extract Class, Extract Subclass, Extract Method

# Bloaters: Long Method



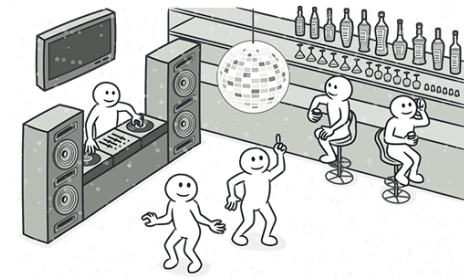
## Signs/Symptoms

- A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- Conditional operators and loops are a good clue that code can be moved to a separate method
- A block of code with a comment that tells you what it is doing can be replaced by a method

## Treatment

- Extract Method

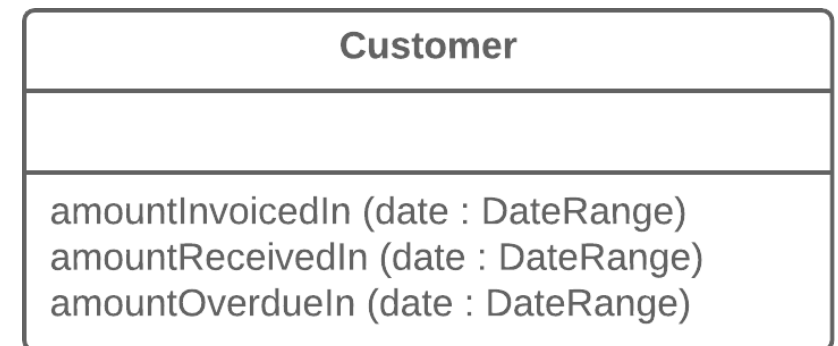
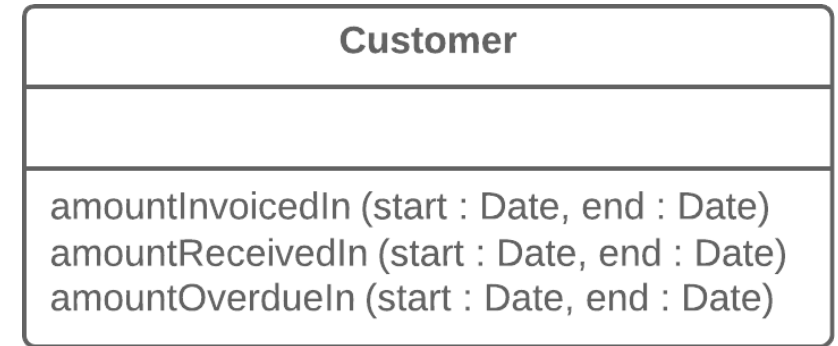
# Bloaters: Data Clumps



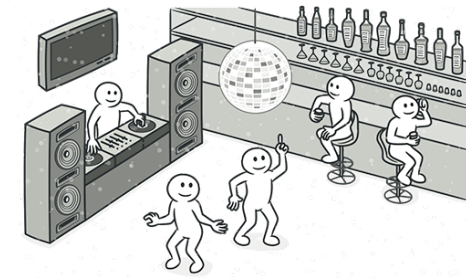
- Sometimes different parts of the code contain identical groups of variables called clumps. (e.g., 3 integers for RGB colors).
- Since these data items are not encapsulated in a class, this increases the sizes of methods and classes.
- To test for this, delete one of the values and see if the others still make sense.

## Treatment

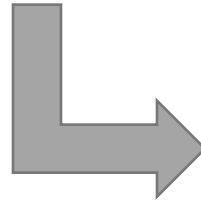
- If these variables are passed as parameters, **replace them with an object**
- **Extract Class**



# Bloaters: Data Clumps

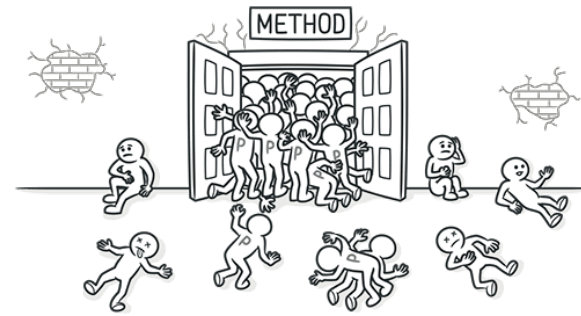


```
class DateUtil {  
    boolean isAfter(int year1, int month1, int day1, int year2, int month2, int day2) {  
        // implementation  
    }  
  
    int differenceInDays(int year1, int month1, int day1, int year2, int month2, int day2) {  
        // implementation  
    }  
  
    // other date methods  
}
```



```
class Date {  
    int year;  
    int month;  
    int day;  
}  
  
class DateUtil {  
    boolean isAfter(Date date1, Date date2) {  
        // implementation  
    }  
  
    int differenceInDays(Date date1, Date date2) {  
        // implementation  
    }  
  
    // other date methods  
}
```

# Bloaters: Long Parameter List



## Signs/Symptoms

- More than three or four parameters for a method
- Hard to understand such lists

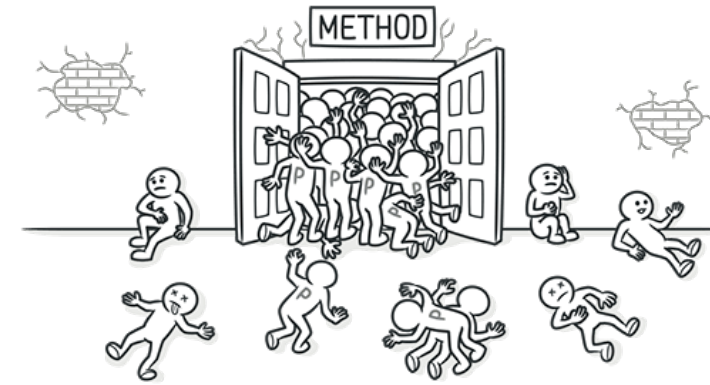
## Reasons

- Several types of algorithm are merged in a single method
- A long list may have been created to control which algorithm (or path) is run

## Treatment

- Breakup the algorithm
- **Replace Parameters with Method Call.**
- Instead of passing a group of data received from another object as parameters, pass the object itself to the method, by using **Preserve Whole Object.**

# Bloaters: Long Parameter List



- You get several values from an object and then pass them as parameters to a method.
- Instead, try passing the whole object.

```
Long parameter lists:  
  
function createUser(name, email, phone, address, city, state,  
country, zipcode) {  
  // ...  
}
```

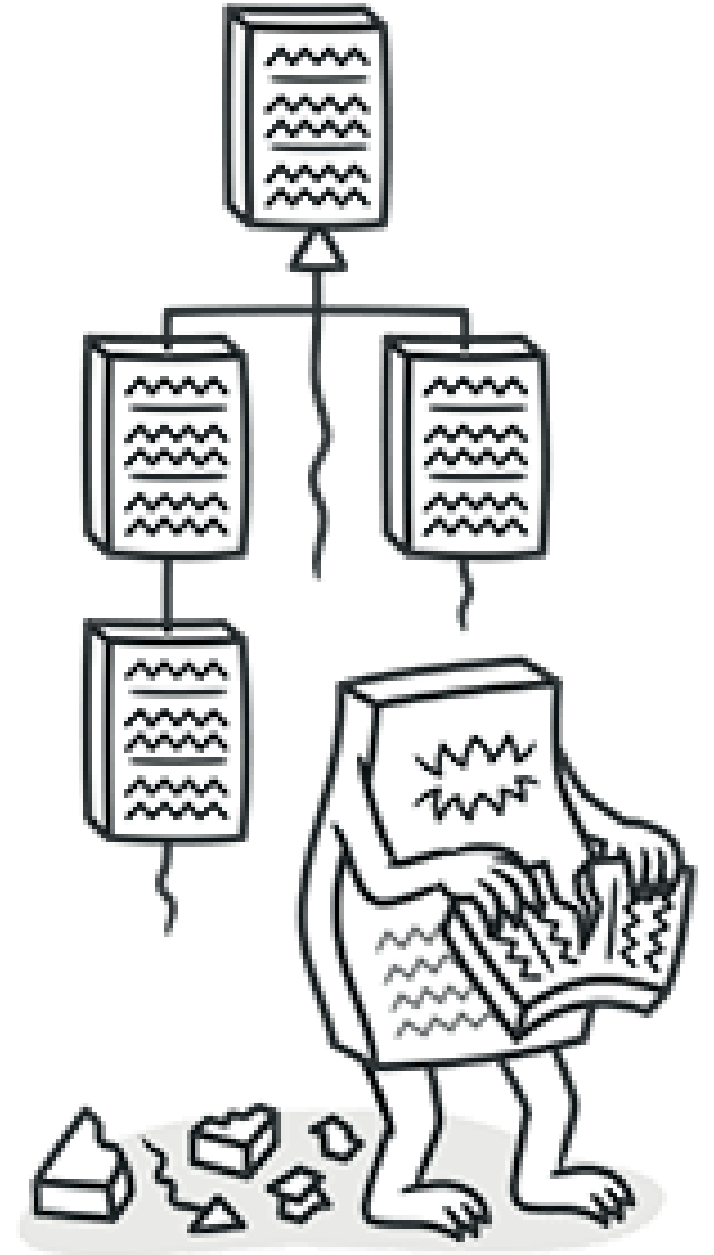
# Summary: Bloaters

Group name	Short description	Smells in group	Treatment
<b>Bloaters</b>	something that has grown so large that it cannot be effectively handled.	<ul style="list-style-type: none"><li>-Long method</li><li>-Large class</li><li>-Long parameter list</li><li>-Data clumps</li></ul>	<ul style="list-style-type: none"><li>-Extract Method</li><li>-Extract class/subclass</li><li>-Replace parameters with method call/preserve object</li><li>- Replace with object/Extract Class</li></ul>

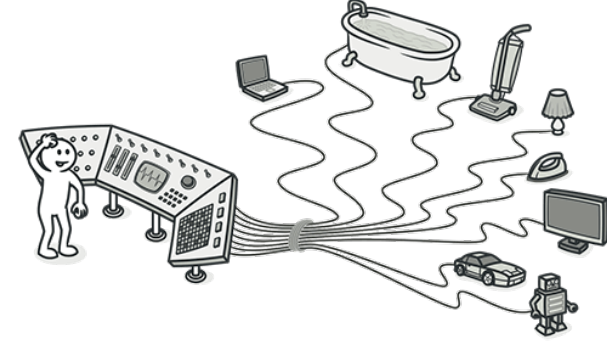


# OO Abusers

Switch Statements, Refused bequest



# OO Abusers: Switch Statements



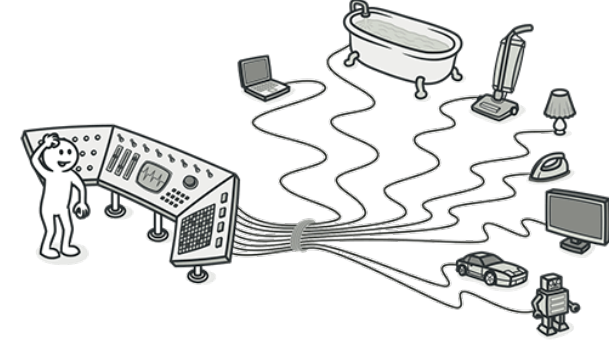
- You have a complex **switch** operator or sequence of **if** statements.

Relatively rare use of **switch** and **case** operators is one of the hallmarks of object-oriented code.

## Treatment

- Extract and Move Method
- Use polymorphism to control implementation (method override)

# OO Abusers: Switch Statements



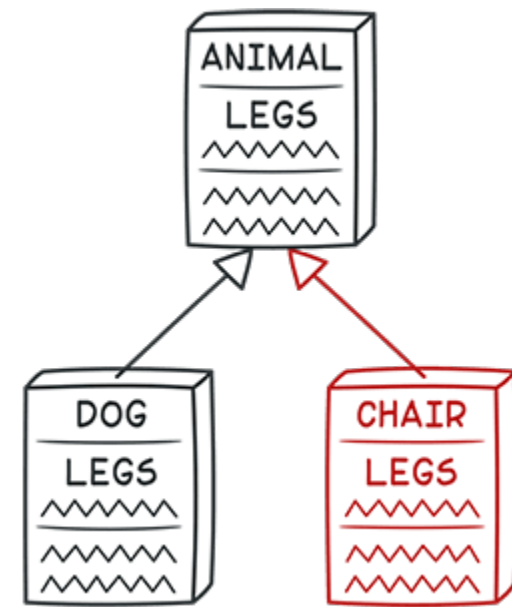
```
class Animal {  
    String type;  
  
    String makeSound() {  
        switch (type) {  
            case "cat":  
                return "meow";  
            case "dog":  
                return "woof";  
            default:  
                throw new IllegalStateException();  
        }  
    }  
}
```



```
abstract class Animal {  
    abstract String makeSound();  
}  
  
class Cat extends Animal {  
    @Override  
    String makeSound() {  
        return "meow";  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    String makeSound() {  
        return "woof";  
    }  
}
```

# OO Abusers: Refused bequest

- Subclasses inherit the methods and data of their parents, but they use only a subset of the implemented parent methods. The unwanted methods may simply go unused or be redefined and throw exceptions
- Possible reason: someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass.
- Also violates the Liskov Substitution Principle.



## Treatment

- **Push down method** - Remove the method or property from Base class and move it to that subclass where it fits.
- Use implementable interfaces
- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, **eliminate inheritance**

# OO Abusers: Refused bequest

Code with Refused Bequest.

```
01. public class Vehicle
02. {
03.     protected void Drive() { }
04. }
05.
06. public class Car : Vehicle
07. {
08. }
09.
10. public class Plane : Vehicle
11. {
12. }
```



Code solving the problem of Refused Bequest.

```
01. public class Vehicle
02. {
03. }
04.
05. public class Car : Vehicle
06. {
07.     void Drive() { }
08. }
09.
10. public class Plane : Vehicle
11. {
12. }
```

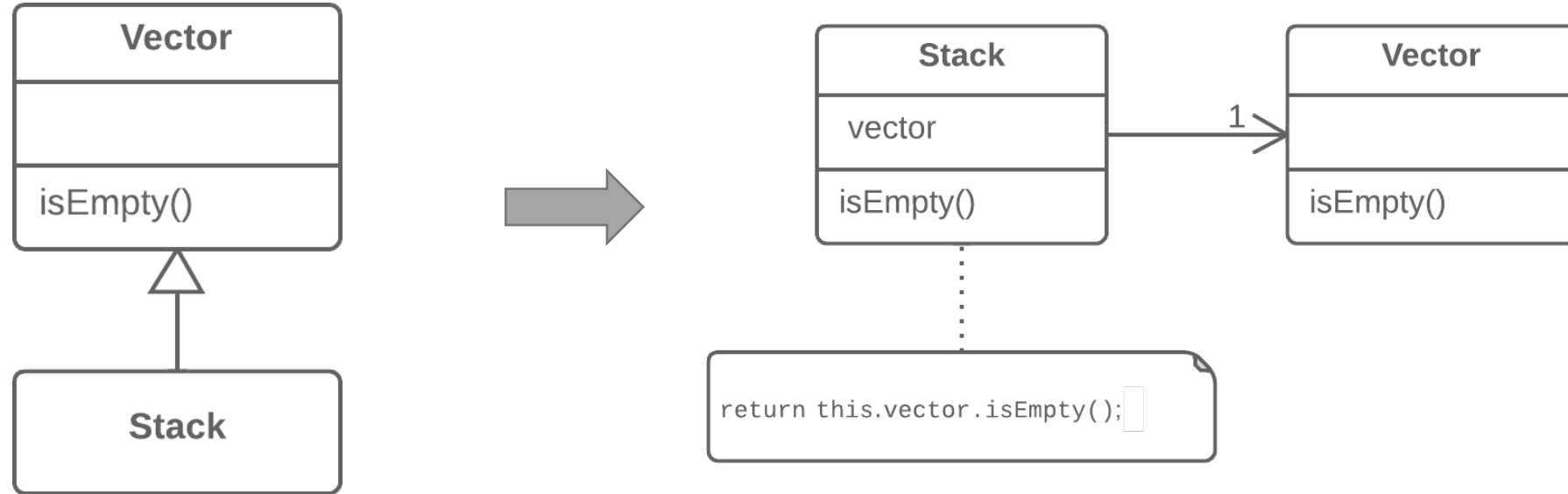
**Push down method** example

# OO Abusers: Refused bequest

## Remove Inheritance Example

You have a subclass that uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data).

Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance.

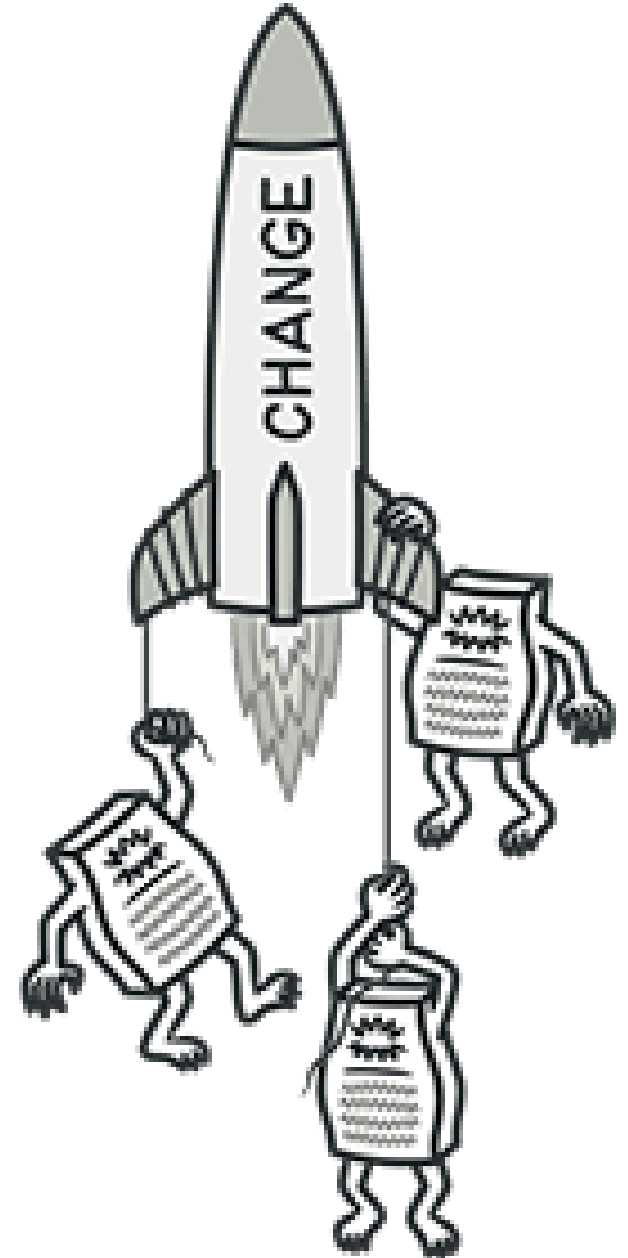


# Summary: OO Abusers

Group name	Short description	Smells in group	Treatment
<b>Object-Orientation Abusers</b>	cases where the solution does not fully exploit the possibilities of object-oriented design.	-Switch statements -Refused Bequest	-Use polymorphism or Extract and move method -Push down method or Eliminate inheritance or implement interfaces

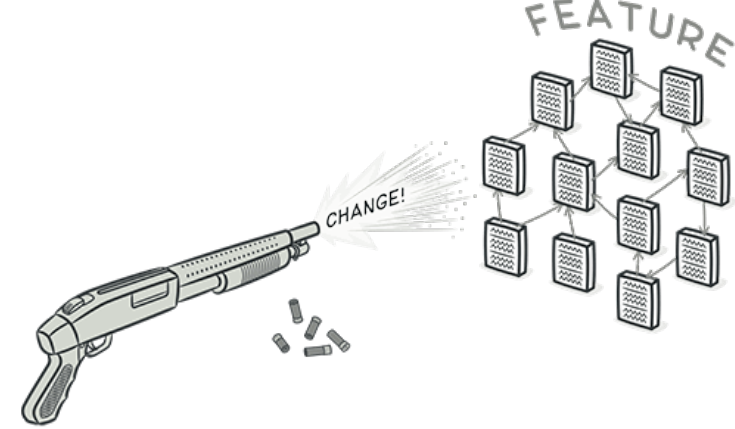
# Change Preventers

Shotgun Surgery & Divergent Change





# Change Preventers: Shotgun surgery

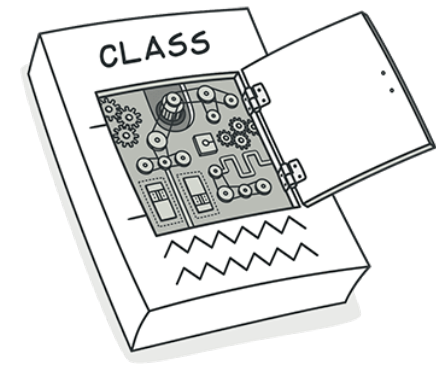


- Every time you make a modification, you must make many small changes to many classes
- Symptom that functionality is spread among classes, you have to change many classes for a small change
- Too much coupling between classes

## Treatment

- move existing class behaviors into a single class. If there's no class appropriate for this, create a new one.

# Change Preventers: Divergent Change



- Resembles **Shotgun Surgery** but is actually the opposite smell
- **Divergent Change** is when many changes are made to a single class. Shotgun Surgery refers to when a single change is made to many classes simultaneously.
- Possible reasons: due to poor program structure or “cypypaste” programming.
- Violates High Cohesion; May violate Single Responsibility Principle

## Treatment

- Extract Class/Method

# Summary: Change Preventers

Group name	Short description	Smells in group	Treatment
<b>Change Preventers</b>	smells that hinder changing or further developing the software.	-Divergent Change -Shotgun surgery	-Extract Class, Method -Move methods and fields

# Dispensables

Duplicated code, Lazy Classes, and Data Classes

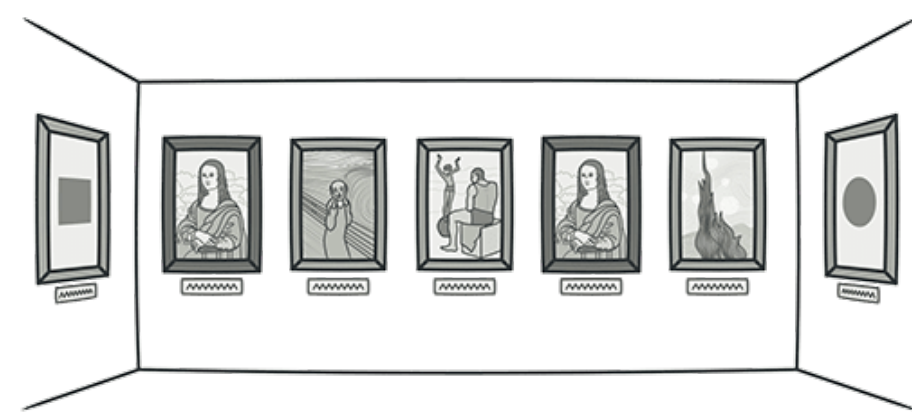


# Dispensables: Duplicated code

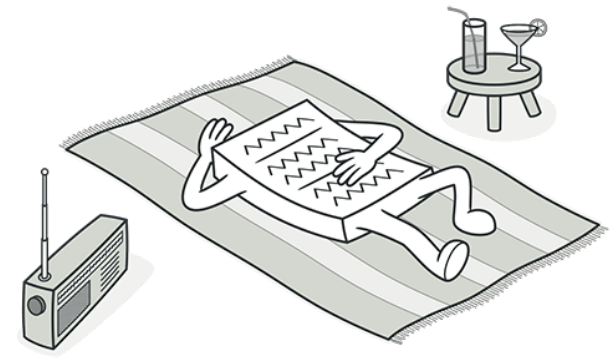
- Most common
- Sections of code repeated all over the place
- Sign of amateur work
- When refactoring duplicated code, you must effectively search for all instances of that code

## Treatment

- Extract Method



# Dispensables: Lazy Class

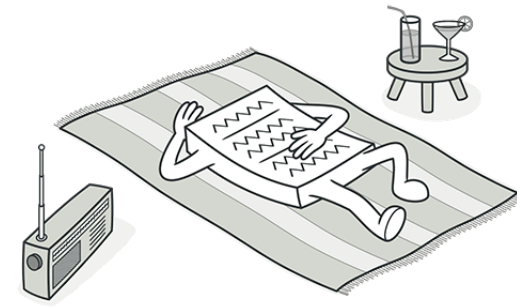


- If a class doesn't do enough to earn your attention, it should be deleted
- Possible reasons:
  - class was designed to be fully functional but after some of the refactoring it has become ridiculously small
  - it was designed to support future development work that never got done

## Treatment

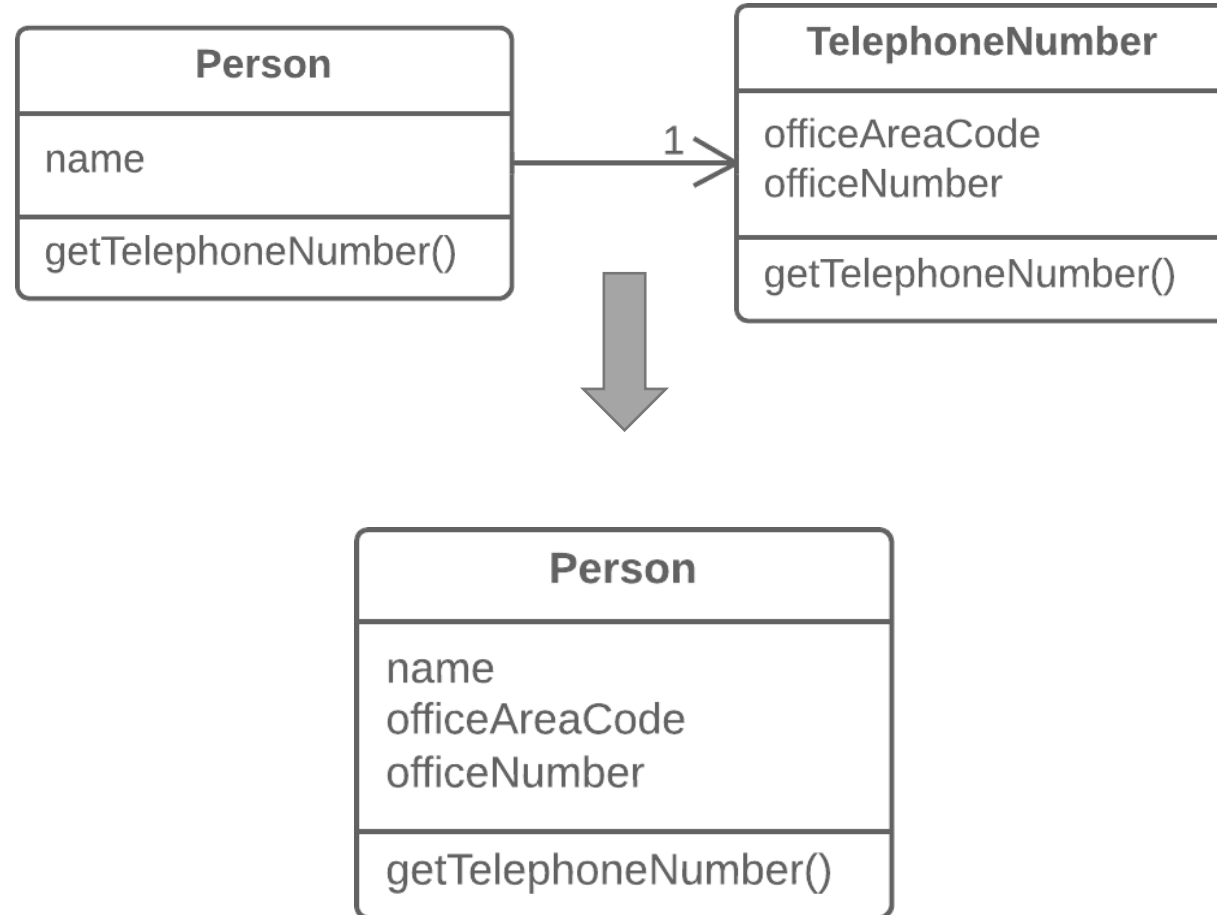
- Inline Class

# Dispensables: Lazy Class

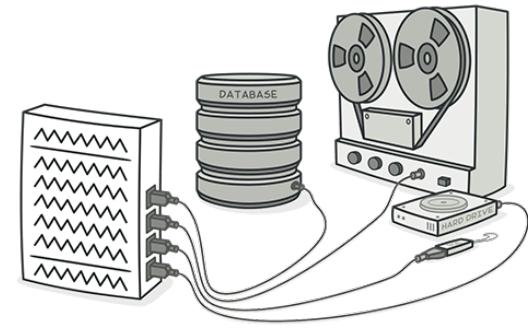


## Inline Class Example

A class does almost nothing and isn't responsible for anything, and no additional responsibilities are planned for it.



# Dispensables: Data Class



- 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 do not contain any additional functionality and can't independently operate on their data.

## Treatment

- Move method and Extract Method to move functionality to the data class
- Encapsulations – to hide from direct access and require that access be performed via getters and setters only
  - Identify methods that operate on the data you're encapsulating and consider moving them to this new class.

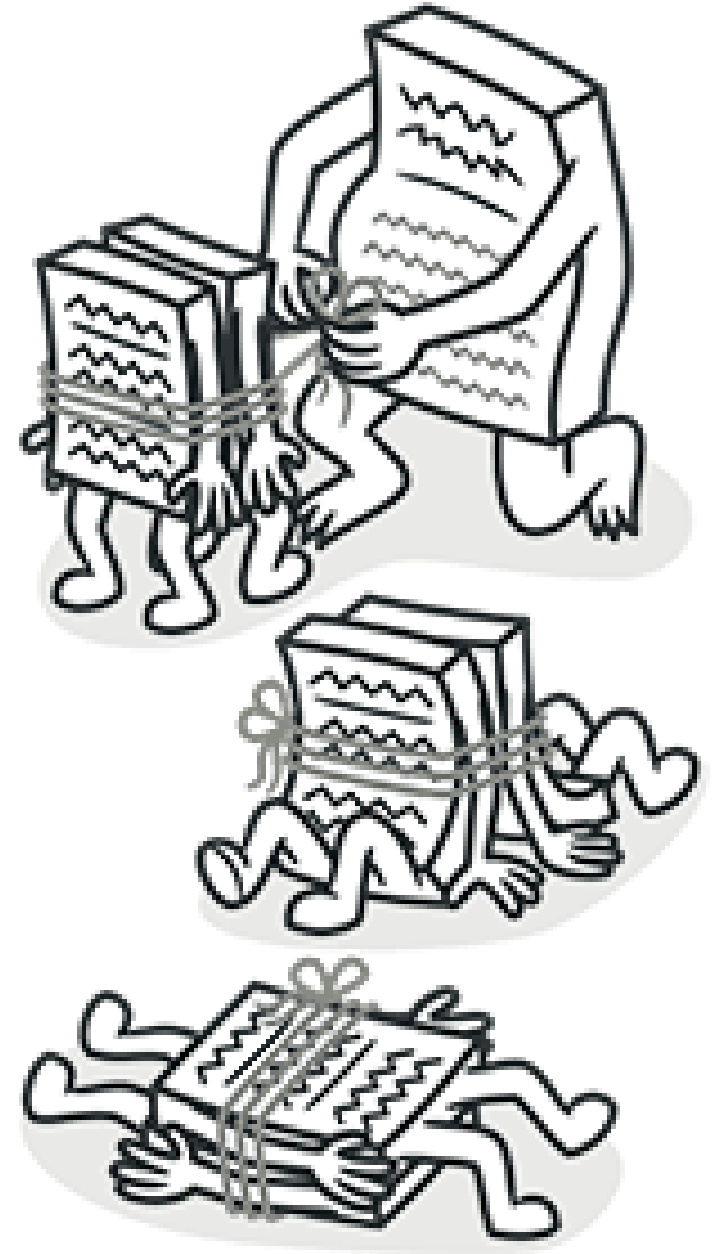


# Summary: Dispensables

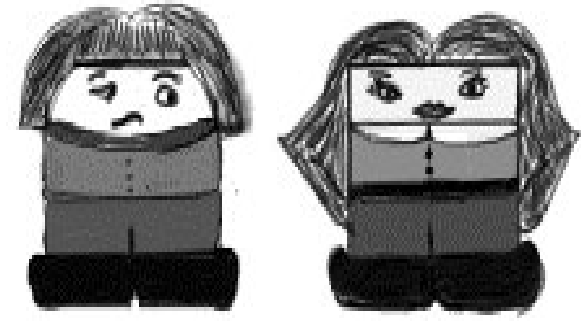
Group name	Short description	Smells in group	Treatment
<b>Dispensables</b>	represent something unnecessary that should be removed from the source code.	-Lazy class -Data class -Duplicated code	-Inline class -Move method/Extract Method/Encapsulation -Extract Method

# Couplers

Feature envy, Inappropriate intimacy, & Middle man



# Couplers: Feature Envy



- A method that seems more interested in a class other than the one it is in
- Most common focus of the envy is the data
- As a basic rule: if things change at the same time, you should keep them in the same place.

## Treatment

- **Move Method:** determine which class has most of the data and put the method with that data

# Couplers: Feature Envy - Example

```
public class User {  
    private ContactInfo contactInfo;  
    public User(ContactInfo contactInfo) { this.contactInfo = contactInfo; }  
    public String getFullAddress() {  
        String streetName = contactInfo.getStreetName();  
        String city = contactInfo.getCity();  
        String state = contactInfo.getState();  
        return streetName + ", " + city + ", " + state;  
    }  
}
```

What is the solution?

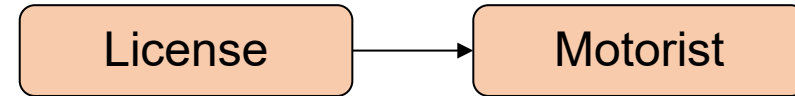
# Couplers: Inappropriate Intimacy

- One class uses the internal fields and methods of another class. Classes know too much about each other
- Violating Low Coupling
- Bi-directional behavior between classes creates tight inter-dependency, i.e., classes are tightly coupled
- Good classes should know as little about each other as possible. Such classes are easier to maintain and reuse.

## Treatment

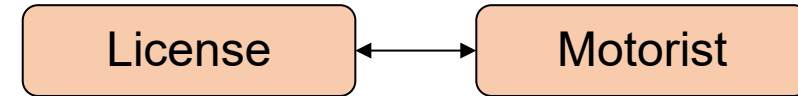
- **Move Method/Move Field:** move parts of one class to the class in which those parts are used. But this works only if the first class truly doesn't need these parts.
- Change Bidirectional Association to Unidirectional

# Couplers: Inappropriate Intimacy Example 1/2



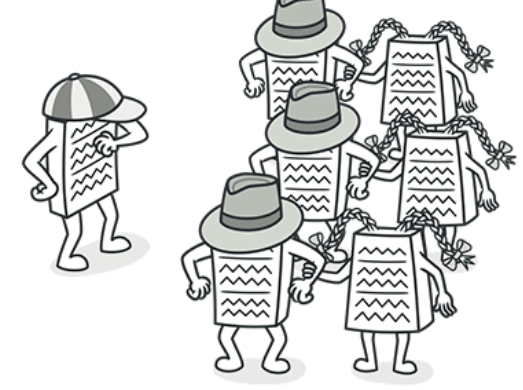
```
public class License {
    private Motorist motorist;
    private int points = 0;
    public void setMotorist(Motorist motorist) { this.motorist = motorist; }
    public int getPoints() { return points; }
    public void addPoints(int points) { this.points += points; }
    public String getSummary() {
        return motorist.getTitle() + " " + motorist.getFirstName()
            + " " + motorist.getLastName() + ", " + getPoints()
            + " points";
    }
}
```

# Couplers: Inappropriate Intimacy Example 2/2



```
public class Motorist {
    private String title;
    private String firstName;
    private String lastName;
    private License license;
    public String getTitle() { return title; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public RiskFactor getRiskFactor() {
        if (license.getPoints() > 3)
            return RiskFactor.HIGH_RISK;
        if (license.getPoints() > 0)
            return RiskFactor.MODERATE_RISK;
        return RiskFactor.LOW_RISK;
    }
}
```

# Couplers: Middle Man



- If a class performs only one action, delegating work to another class, why does it exist at all?
- Violating SRP
- Possible reasons: it can be the result of the useful work of a class being gradually moved to other classes. The class remains as an empty shell that doesn't do anything other than delegate.

## Treatment

- Inline Class



# Summary: Couplers

Group name	Short description	Smells in group	Treatment
<b>Couplers</b>	a measure of how closely connected two routines or modules are	-Feature envy -Inappropriate intimacy -Middle man	-Move method/field -Move methods, Bidirectional to unidirectional association -Inline Class

# Summary

Group name	Short description	Smells in group	Treatment
<b>Bloaters</b>	something that has grown so large that it cannot be effectively handled.	<ul style="list-style-type: none"><li>-Long method</li><li>-Large class</li><li>-Long parameter list</li><li>-Data clumps</li></ul>	<ul style="list-style-type: none"><li>-Extract Method</li><li>-Extract class</li><li>-Replace parameters with method call/object</li><li>- Preserve whole object</li></ul>
<b>Object-Orientation Abusers</b>	cases where the solution does not fully exploit the possibilities of object-oriented design.	<ul style="list-style-type: none"><li>-Switch statements</li><li>-Refused Bequest</li></ul>	<ul style="list-style-type: none"><li>-Use polymorphism</li><li>-Push down method</li><li>-Eliminate inheritance</li></ul>
<b>Change Preventers</b>	smells that hinder changing or further developing the software.	<ul style="list-style-type: none"><li>-Divergent Change</li><li>-Shotgun surgery</li></ul>	<ul style="list-style-type: none"><li>-Extract Class, Method</li><li>-Move methods and fields</li></ul>
<b>Dispensables</b>	represent something unnecessary that should be removed from the source code.	<ul style="list-style-type: none"><li>-Lazy class</li><li>-Data class</li><li>-Duplicated code</li></ul>	<ul style="list-style-type: none"><li>-Inline class</li><li>-Move method/Extract Method/Encapsulation</li><li>-Extract Method</li></ul>
<b>Couplers</b>	a measure of how closely connected two routines or modules are	<ul style="list-style-type: none"><li>-Feature envy</li><li>-Inappropriate intimacy</li><li>-Middle man</li></ul>	<ul style="list-style-type: none"><li>-Move method/field</li><li>-Bidirectional to unidirectional association</li><li>-Inline Class</li></ul>

# Top 10 Code Smells to Identify in PRs

<https://axolo.co/blog/p/top-10-code-smells-to-identify-in-pull-requests-with-code-examples>